Un modèle de réécriture pour l'intégration de services

Olivier Nano

THÈSE DE DOCTORAT

École doctorale « Sciences et Technologies de l'Information
et de la Communication » de Nice - Sophia Antipolis
Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES ET TECHNIQUES

UN MODÈLE DE RÉÉCRITURE POUR L'INTÉGRATION DE SERVICES

---

présentée et soutenue publiquement par

Olivier NANO

Thèse dirigée par

Mireille BLAY-FORNARINO et Michel RIVEILL

Soutenue le 6 décembre 2004 *à* l'E.S.S.I. *devant le jury composé de*

Président du Jury	Paul FRANCHI	Université de Nice - Sophia Antipolis
Rapporteurs	Laurence DUCHIEN	Université de Lille - LIFL
	Jacques NOYÉ	École des Mines de Nantes
	Christian QUEINNEC	Université de Paris 6 - LIP6
Examinateur	Mark GILBERT	Microsoft EMIC

À mes parents,
ma famille,
mes amis.

— Olivier.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

Remerciements

Je voudrais tout d'abord remercier les membres du jury, Laurence Duchien, Mark Gilbert, Jacques Noyé, Christian Queinnec et Paul Franchi pour m'avoir fait l'honneur d'évaluer mes travaux de recherche et les rapporteurs, Laurence Duchien, Jacques Noyé et Christian Queinnec pour leurs remarques pertinentes.

Mes directeurs de thèse Mireille Blay-Fornarino et Michel Riveill pour leur soutien, leurs conseils, leur patience, leur gentillesse, leur temps, et cet amour de la recherche.

L'équipe Rainbow : Anne-Marie, Audrey, David, Jean-Yves, Jeremy, Karima, Michel, Mireille, pour leur accueil et cette vie d'équipe si agréable et chaleureuse.

Stéphane Ducasse pour ses cours, ses conseils, Smalltalk, toutes ces discussions.

Tous les habitués de #superdede : Breton, DeM, GianCarlo, Looz, Mkris, Nini, Nono, Udab pour toutes ces discussions enflammées, ces parties réseau, ces déjeuners partagés.

Les musiciens de Gust : Benji, Damien, David et Laetitia pour tous ces moments de plaisir musicale et d'amitié.

Tous mes amis (dans le désordre) : Christine, Nico, Doth, Coz, Pierre, Pierre-M, Tony, Julien, Aurelie, Marie, Toto, Sylvie, Lou, Thom, Karim, JR, Fred, Sonia, Brice, Seb, Gwen, Marc, Lolo, Johana, Jeremy, Nat, Maud, Olive, Vince, Lolo, Christian, Tif, Christel, Nico, Greg, Jean-Raph, Steph pour tout le bonheur que vous m'apportez au quotidien.

Mes parents et ma famille : Gérard, Elena, Ghislaine, Julie, Théo-Paul, Eric, Papy, Chantal, Georges, Carine, Rosette, Henri, Nina, Robert, Claude, Alain, Steph, Claire, Karine, Christophe, Katia, Robert, Yelena, Lukas pour l'amour et le bien être dans lequel je grandis.

Tous les membres de l'EMIC pour leur accueil chaleureux et ce départ pour de nouvelles aventures !

Merci.
— Olivier.

CHAPITRE 1

Introduction générale

L'objectif de cette thèse est de proposer un modèle pour l'intégration de services qui soit indépendant des plateformes à composants, qui permette de séparer la conception des services de leur intégration et qui offre des propriétés de composition et de détection d'erreurs. Dans ce chapitre, nous motivons tout d'abord l'intérêt des travaux présentés dans cette thèse et nous en définissons clairement le contexte. Nous poursuivons par une présentation de nos objectifs et nous posons le vocabulaire employé en donnant une définition précise de la notion de « services », « d'intégration de services » et de « plateforme à composants ». Nous décrivons finalement le plan de cette thèse.

Cette thèse a donné lieu à des publications [8, 62, 63, 64, 65, 66] dans les domaines des plateformes à composants et de l'ingénierie dirigée par les modèles.

1.1 Problématique initiale

Les systèmes informatiques se complexifient, les temps accordés au développement tendent à se raccourcir. Les points clés du développement logiciel sont maintenant la réutilisation et l'assemblage. Un nouveau rôle émerge : le configurateur d'application. Il doit assembler et configurer les applications à partir des briques logicielles fournies par les développeurs.

Pour aider les développeurs à construire des briques logicielles, la programmation par composants propose d'accroître la modularité et la réutilisation du code : premièrement par l'utilisation d'interfaces et de protocoles de communication pour isoler les composants les uns des autres, deuxièmement par la séparation du code métier et du code des services (aussi nommé code technique) tels que l'authentification, les transactions, la persistance, l'audit, la trace, la récupération d'erreurs, la répartition de charge, etc. La diffusion des intergiciels de communication comme CORBA ont montré que la gestion du code technique était répétitive et complexe. C'est sur cette base que les plateformes à composants automatisent la gestion et l'intégration du code technique.

La programmation par composants permet de répartir les tâches de la construction d'une application entre plusieurs acteurs [98]. Le *développeur* de composants code la partie métier du composant en respectant la spécification de la plateforme

à composants (Enterprise JavaBean [103], CORBA Component Model [84], Fractal [27], .Net[61]) dans laquelle le composant devra s'exécuter. Ensuite l'*assembleur* de l'application assemble les composants qui définissent l'application. Le *déployeur* de l'application intègre à l'application les services requis et déploie l'application finale sur le site. Généralement l'intégration des services dans l'application se fait de manière déclarative à l'aide de fichiers de déploiement qui spécifient la politique des services à intégrer à l'application. Pour répondre à un besoin croissant de nouveaux services dans les applications, un quatrième acteur vient compléter cette description. Le *fournisseur de services* intègre de nouveaux services dans les plateformes à composants et les met à disposition des applications (cf. figure 1.1). Comme nous allons le voir, dans certaines approches, le fournisseur de services et le fournisseur de plateformes sont confondus.

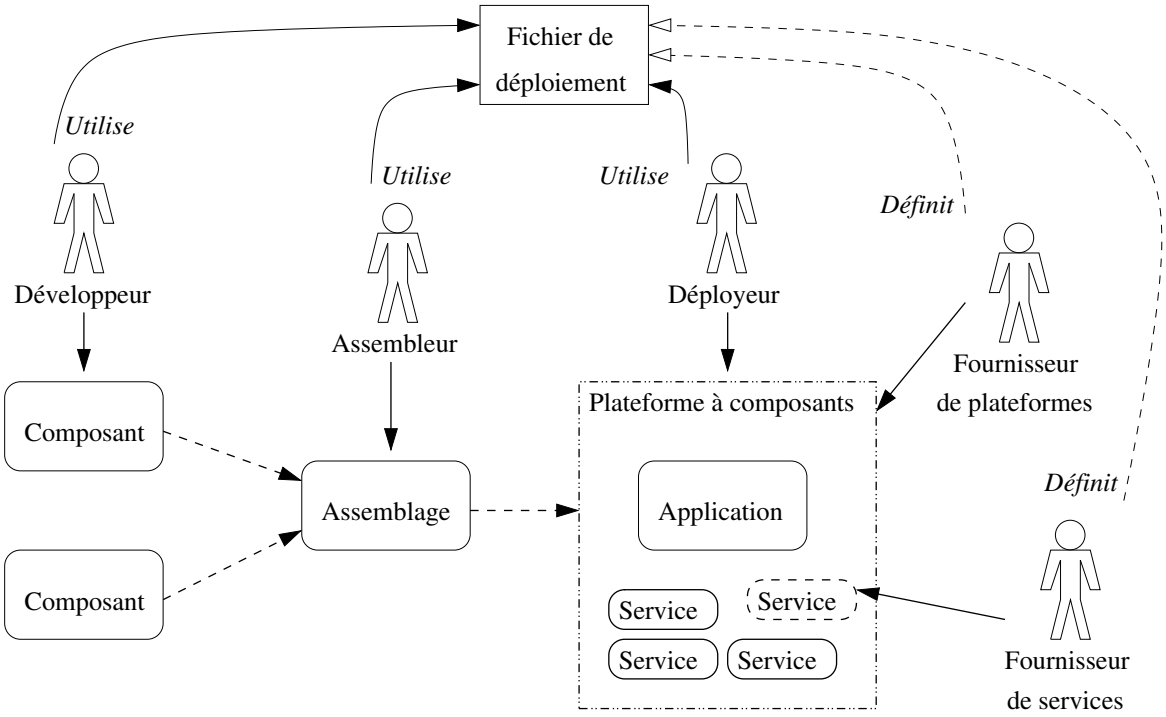

FIG. 1.1 – Acteurs du monde des composants

L'intégration de services dans une application est effectuée automatiquement par la plateforme à composants à partir des déclarations du déployeur. Ces déclarations sont regroupées dans le fichier de déploiement. Ce fichier de déploiement permet entre autre de configurer et paramétrer les services. L'adaptation du service pour les besoins précis du déployeur passe uniquement par les paramètres du fichier de déploiement.

Les implantations de plateformes à composants définissent une infrastructure qui va contrôler l'exécution des composants et leur offrir les services dont ils ont besoin. Les objets d'interposition tels que les adaptateurs, proxy, conteneurs, interceptent les requêtes destinées aux composants. Ce sont ces objets d'interposition qui gèrent les services pour l'application. Chaque implantation de plateforme à composants propose sa propre infrastructure. L'hétérogénéité des infrastructures rend la diffusion de nouveaux services difficile. Pour ajouter un nouveau service dans une plateforme,

le fournisseur de services doit connaître les détails de l'infrastructure pour décider où (dans quels objets d'interposition) insérer les appels vers le nouveau service, et cela pour chaque plateforme à composants dans laquelle il veut ajouter son nouveau service. Les objets d'interposition étant générés automatiquement par la plateforme, le fournisseur de services doit modifier le code du générateur pour que celui-ci génère des objets d'interposition qui offrent le nouveau service aux applications.

L'intégration de plusieurs services dans une application implique de les composer entre eux. Déterminer l'ordre d'exécution des services, pour que le résultat de la composition soit sémantiquement correct vis-à-vis de l'application, est complexe. Le fournisseur de services doit donc gérer de manière attentive l'intégration d'un service par rapport aux autres services qu'utilise l'application et donc de manière générale vis-à-vis de toutes les compositions potentielles des services offerts par la plateforme.

1.2 Objectif et approche choisie

L'objectif de cette thèse est d'offrir au fournisseur de services un support pour l'intégration de services dans les plateformes à composants. Ce support de l'intégration de services doit permettre de résoudre le problème de l'hétérogénéité qui inclut la difficulté de modifier les générateurs et le problème de la composition des services.

L'approche générale que nous avons choisie pour résoudre ces problèmes se décompose en trois phases :

- Décrire l'intégration des services au niveau d'un modèle indépendant des plateformes à composants.
- Composer automatiquement les intégrations de services dans ce modèle indépendamment de l'ordre des déclarations d'intégration de services ce qui permet une approche séparée des différents aspects d'une application et une détection des conflits.
- Projeter ces descriptions d'intégration de services dans les plateformes à composants.

Nous définissons un modèle de descriptions abstraites de composants qui représente des composants de manière abstraite indépendamment des plateformes à composants. Ce modèle permet de s'abstraire des plateformes à composants et définit des opérations pour l'évolution structurelle et comportementale des composants. Pour permettre de manipuler ce modèle et permettre le paramétrage de l'intégration de services, nous définissons un modèle qui précise les modalités de l'intégration (nommé intégrateur).

Nous utilisons un mécanisme de composition automatique qui permet de fusionner de manière commutative et associative les descriptions d'intégration des différents services. Ce mécanisme de composition repose, d'une part sur un système de réécriture pour la composition des évolutions comportementales, et d'autre part sur l'utilisation d'opérations ensemblistes pour la composition des évolutions structurelles. Ce mécanisme permet de détecter des incompatibilités entre les intégrations de services

La projection dans les plateformes à composants se fait en passant par le modèle intermédiaire des composants concrets qui permet de combiner les évolutions des intégrations de services et des informations issues de la plateforme à composants cible. La projection permet de détecter des incompatibilités entre les intégrations de services et la plateforme cible. Enfin le code final est généré dans la plateforme cible.

1.3 Vocabulaire

Dans cette thèse nous employons le vocabulaire de la programmation par composants. Nous reprenons en particulier les principales définitions de Clemens Szyperski dans « *Component software : beyond object-oriented programming* » [105] :

Composant : désigne une unité de composition avec des interfaces et des dépendances explicites. Un composant est une unité déployable.

Nous différencions dans les plateformes à composants la spécification de l'implantation :

Plateforme à composants : désigne la spécification d'une infrastructure de support aux composants (comme par exemple la spécification EJB, CCM, Fractal, .net).

Implantation de plateforme à composants : désigne une implantation particulière de plateforme à composants (comme par exemple Jonas [69] pour la spécification EJB, OpenCCM [70] pour la spécification CCM, Julia [68] pour la spécification Fractal, mono [67] pour la spécification .net).

Définition d'un *service* et de l'*intégration de services* :

Service : désigne du code technique par opposition au code métier. Un service est un ensemble de composants qui servent à l'exécution des composants métier.

Intégration de services : désigne la modification d'une application ou de sa plateforme d'exécution pour y ajouter des services. L'intégration de services dans des composants consiste à ajouter des liaisons entre des composants de service et des composants métiers. Nous parlerons plus généralement d'intégration de services.

1.4 Plan du manuscrit

Cette thèse se découpe en trois parties.

La première partie étudie différentes approches de l'intégration de services et expose la problématique de cette thèse. Dans un premier temps nous discutons de l'intégration de services dans les intergiciels de communication. Nous montrons que l'intégration manuelle d'un service est une opération répétitive et complexe. Dans un deuxième temps nous étudions les plateformes à composants qui gèrent les services pour l'application. Nous discutons de l'hétérogénéité des plateformes, de la difficulté pour le fournisseur de services d'intégrer de nouveaux services dans les différentes plateformes à composants puisqu'il est obligé de modifier les générateurs de code pour réaliser cette intégration, et de la difficulté de composer les services manuellement. Dans un troisième temps nous nous intéressons aux outils qui permettent de faire de la séparation des préoccupations dans le but d'aider le fournisseur de services à intégrer de nouveaux services dans les plateformes à composants de manière plus homogène. Pour cela nous étudions la métaprogrammation, les approches par tissage et les approches par modélisation qui nous permettent d'extraire des concepts important pour notre approche de l'intégration de services. Nous montrons que les deux premières approches sont trop proche du modèle objet et ne permettent pas la prise en compte de la notion de composants, et nous montrons que la troisième approche

utilise des systèmes de transformations ad hoc qui ne sont pas assez proche des implantations des plateformes à composants.

La deuxième partie de cette thèse décrit notre modèle de l'intégration de services. Tout d'abord nous présentons notre approche sous forme d'un scénario et montrons comment cette proposition répond aux problèmes de l'hétérogénéité et de la difficulté de modifier des générateurs ainsi qu'à la composition des services. Ensuite nous décrivons notre modèle de descriptions abstraites de composants et celui des descriptions d'intégration de services. Puis nous décrivons et formalisons le processus de composition des descriptions d'intégration de services. Enfin nous discutons du processus de projections vers les plateformes à composants.

La troisième partie présente des exemples d'intégration de services à l'aide de notre modèle. Dans un premier temps nous décrivons les systèmes externes que nous utilisons dans le cadre des plateformes à composant EJB, CCM et Fractal. Puis nous définissons l'intégration de services de cache de méthodes, d'authentification, de cryptage et de persistance dans une application de calcul.

Finalement nous concluons et donnons les perspectives à ce travail.

Première partie

ÉTUDE DE L'EXISTANT : SUPPORT DE L'INTÉGRATION DE SERVICES

Dans cette partie...

Dans cette partie nous étudions le support de l'intégration de services dans différents modèles de programmation. Dans l'introduction (cf. chapitre 1) nous avons défini des propriétés qui facilitent l'intégration de services pour le fournisseur de services ainsi que pour l'utilisateur de services. Nous évaluons l'intégration de services dans les différents modèles de programmation suivant les cinq propriétés que sont :

La gestion de l'hétérogénéité : est-ce que l'approche permet au fournisseur de services de porter ses définitions d'intégration de services dans différentes systèmes équivalents ?

La séparation des définitions de l'intégration de services : est-ce que les définitions des intégrations de services sont indépendantes les unes des autres ?

L'adaptabilité de l'intégration de services : quels sont les moyens de l'utilisateur pour adapter le service à son application ?

Le découplage du fournisseur et de l'utilisateur : quelles sont les informations qui transitent entre le fournisseur et l'utilisateur ?

La complexité de mise en œuvre : est-ce que l'approche est complexe à mettre en œuvre du point de vue du fournisseur ? de l'utilisateur ?

Nous commençons par l'étude de l'intégration de services dans les intergiciels de communication (cf. chapitre 2). Nous prenons en exemple CORBA 2.0 [75] qui nous permet de discuter des services normalisés par l'OMG (Object Management Group). Ce chapitre souligne les modifications à apporter à une application pour y intégrer des services et nous montre que l'intégration d'un service est une opération répétitive et complexe.

Puis nous examinons l'intégration de services dans les plateformes à composants (cf. chapitre 3). Nous étudions les différents modèles que sont les EJB [103], les CCM [84] et Fractal [27], et quelques unes de leurs implantations. Ce chapitre nous montre que les infrastructures des implantations de plateformes à composants sont hétérogènes ; qu'elles ne facilitent pas l'intégration de nouveaux services et qu'il faut nous munir d'outils pour faciliter l'intégration de services dans les plateformes à composants.

Nous expérimentons ensuite différents outils qui permettent de faire de la séparation des préoccupations (cf. chapitre 4). Nous évaluons ces outils toujours dans

le cadre de l'intégration de services dans les plateformes à composants. Nous étudions les systèmes réflexifs qui supportent la métaprogrammation (comme CLOS [9] et CODA [2] par exemple), puis les approches par tissage (comme la programmation par aspects [49] et les langages dédiés [108]) et enfin les approches de l'intégration de services par modélisation (comme la modélisation orientée sujet et UML 2.0 [79]). Ce chapitre nous permet de déduire les éléments clés qui vont nous permettre de développer un modèle de l'intégration de services qui réponde aux cinq propriétés définies plus haut.

CHAPITRE 2

Intégration de services dans les intergiciels de communication

Dans ce chapitre nous étudions l'intégration de services dans les intergiciels de communication. Ce chapitre nous permet d'introduire les services les plus couramment utilisés que sont la notification, le nommage, la persistance, la sécurité et les transactions. L'étude de ces services et de leur intégration nous permet d'analyser le travail du fournisseur de services ainsi que celui de l'utilisateur de services.

Les intergiciels de communication permettent de s'abstraire de la couche réseau dans les applications reparties. Le développeur d'applications se concentre sur le code métier de son application, et l'intergiciel se charge de la communication entre les entités distantes. Nous prenons comme exemple l'intergiciel de communication CORBA qui a standardisé la notion de service.

CORBA dans ses versions 1 et 2 permet à des applications distribuées écrites dans différents langages de programmation et s'exécutant sur des systèmes d'exploitation différents d'interagir. CORBA enlève de la charge du programmeur les problèmes de communication entre les entités distribuées. Nous rappelons dans un premier temps les principes de fonctionnement de CORBA, puis nous détaillons plusieurs services et leur intégration dans CORBA.

2.1 Les éléments standards de l'architecture CORBA 2.0

CORBA est essentiellement composé de trois concepts (cf. figure 2.1) : un ensemble d'interfaces d'invocations, l'ORB (*Object Request Broker*), et un ensemble d'objets d'adaptations (*POA* - Portable Object Adapter). Les interfaces d'invocation permettent l'empaquetage de données. L'ORB localise le destinataire, la méthode invoquée et transmet les arguments. Les objets d'adaptations dépaquettent les arguments et invoquent la méthode requise sur l'objet destinataire. Les interfaces des objets sont décrites dans un langage commun (OMG IDL) qui permet la construction d'un mécanisme d'empaquetage / dépaquetage générique.

L'ORB est essentiellement un service d'invocation de procédure distante (RPC). L'OMG propose au dessus de cela une architecture de gestion des objets (OMA). On y trouve la définition d'un ensemble d'objets de services : les *CORBAServices*, un ensemble d'objets utilitaires : les *CORBAFacilities* et un ensemble de définitions d'objets d'applications. Les objets de services se concentrent sur les blocs fondamentaux de la

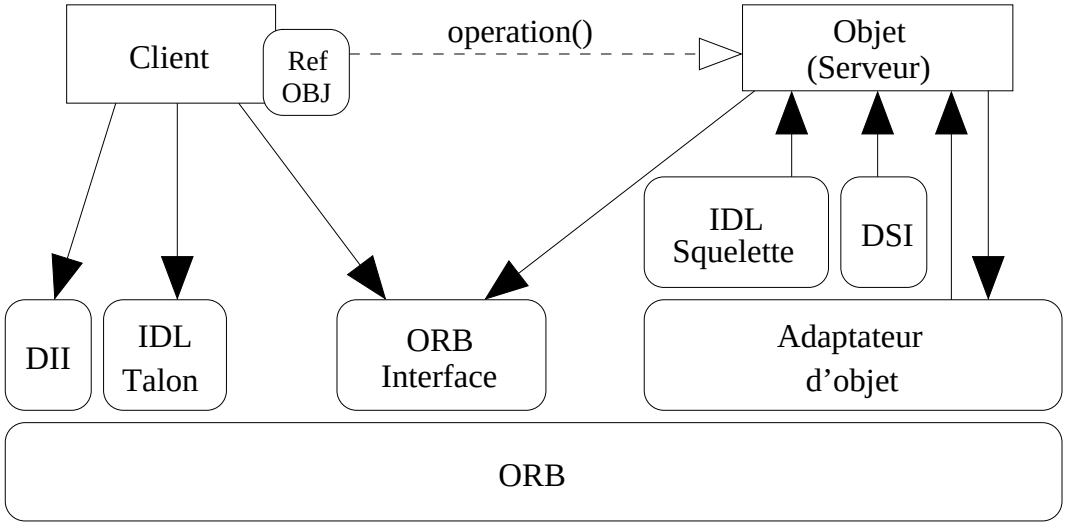

FIG. 2.1 – Architecture de CORBA 2.0

programmation distribuée telle que la diffusion d'événements, les transactions, ou le nommage.

2.2 Les services de l'architecture OMA

Les CORBAServices définissent actuellement 16 objets de services. Nous résumons les services les plus courants (détaillés dans [101]) et nous donnons des détails d'implantations qui nous permettent de discuter de l'intégration de ces services dans des applications CORBA. Les services que nous avons choisis d'examiner sont ceux que l'on retrouve dans la plupart des plateformes à composants (notification, nommage, persistance, sécurité et transactions). Nous discutons des modifications de l'ORB et des modifications des applications qu'il faut effectuer pour réaliser l'intégration de ces services.

2.2.1 Le service d'événements et de notification

Le service d'événements et de notification permet aux objets de communiquer entre eux par envoi d'événements [77, 85]. Les événements vont d'un émetteur à un receveur. Ils circulent à travers des canaux qui séparent l'émetteur du receveur sans que ceux-ci ne se connaissent. Les canaux supportent entre autres les modèles *push* et *pull*. Dans le modèle *push* l'émetteur appelle une méthode *push* sur le canal et tous les receveurs sont notifiés (cf. figure 2.2). A l'inverse dans le modèle *pull* le receveur appelle une méthode *pull* sur le canal pour récupérer des messages s'il y en a (cf. figure 2.3). Ce sont les émetteurs qui décident quand poster des messages.

Bâti sur la spécification du service d'événements, le service de notification ajoute de la qualité de service, l'administration des canaux, des types standard et structurés d'événements, du filtrage dynamique basé sur les types et la qualité de services, etc.

Pour le développeur de services l'implantation du service d'événements ne nécessite pas de modification de l'ORB. Le service est implémenté comme un servant. Au démarrage le servant doit s'enregistrer auprès du serveur de nommage pour être

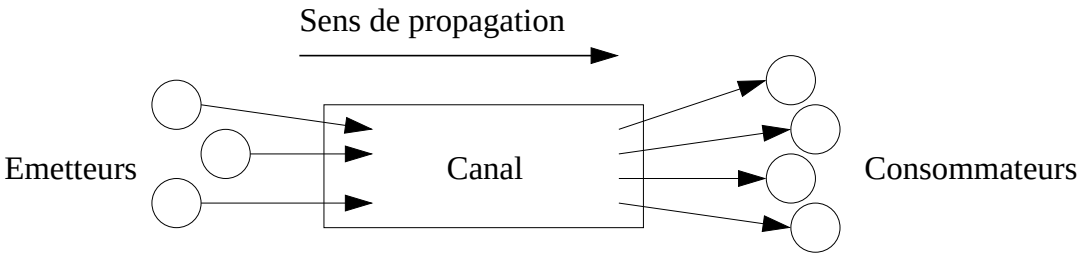

FIG. 2.2 – Service d'événements : le modèle PUSH

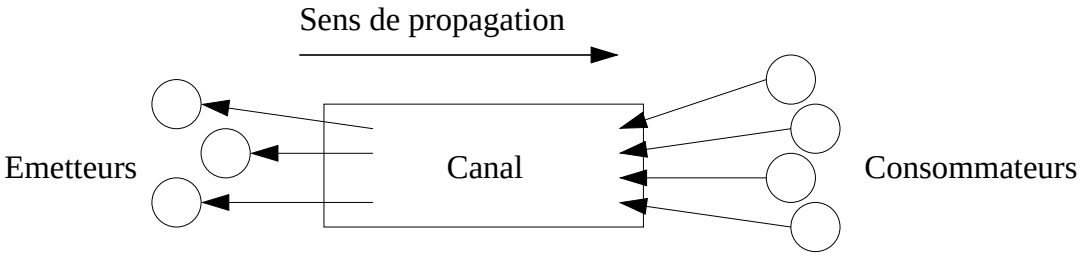

FIG. 2.3 – Service d'événements : le modèle PULL

accessible aux clients du service. Le travail du développeur de service est donc de construire le composant du service, c'est à dire le serveur d'événements. L'implantation d'un tel service est détaillée dans [102].

L'utilisateur de services doit d'abord modifier son code métier pour insérer des appels qui vont lui permettre de se connecter au serveur d'événements, récupérer les références qui vont lui permettre de communiquer, puis enfin placer les appels vers le serveur d'événements pour communiquer. Le détail des opérations à réaliser sur le code métier est décrit dans les tables 2.1 et 2.2. La table 2.1 décrit les opérations à implémenter pour le fournisseur d'événements dans un modèle *push*, et la table 2.2 décrit les opérations à implémenter pour le consommateur d'événements toujours pour le modèle *push*. De plus l'utilisateur doit lancer le service d'événements avant de démarrer son application.

Etapes	**Fournisseur PUSH**
Récupérer une référence sur le serveur d'événements	`EventChannelHelper.narrow( orb.resolve_initial_references( "EventService"))`
Récupérer un administrateur de fournisseur	*EventChannel : :for_suppliers()*
Récupérer un proxy de consommateur	*SupplierAdmin : :obtain_push_consumer()*
Ajouter le fournisseur à l'EventChannel	*ProxyPushConsumer : :connect_push_supplier()*
Transférer des données	*ProxyPushConsumer : :push(in any data)*

TAB. 2.1 – Connecter un fournisseur à un canal d'événements

Etapes	**Consommateur PUSH**
Récupérer une référence sur le serveur d'événements	`EventChannelHelper.narrow( orb.resolve_initial_references( "EventService"))`
Récupérer un administrateur de consommateur	*EventChannel : :for_consumers()*
Récupérer un proxy de fournisseur	*ConsumerAdmin : :obtain_push_supplier()*
Ajouter le consommateur à l'EventChannel	*ProxyPushSupplier : :connect_push_consumer()*
Transférer des données	*Implémenter push()*

TAB. 2.2 – Connecter un consommateur à un canal d'événements

2.2.2 Le service de nommage et le service de trading

Le service de nommage [80] et le service de trading [83] permettent de récupérer l'adresse d'un objet. Le service de nommage permet d'associer un nom quelconque à un objet. Les noms sont uniques dans un contexte de nommage et les contextes de nommage forment une hiérarchie (cf fig. 2.4). Pour ajouter une association objet/référence dans le service de nommage on utilise l'opération `bind`. On supprime cette association par l'opération `unbind`. La création de la hiérarchie des noms se fait grâce au contexte de nommage par l'opération `bind_context`. La récupération d'une référence se fait en invoquant l'opération `resolve` en passant le nom de l'objet au service de nommage.

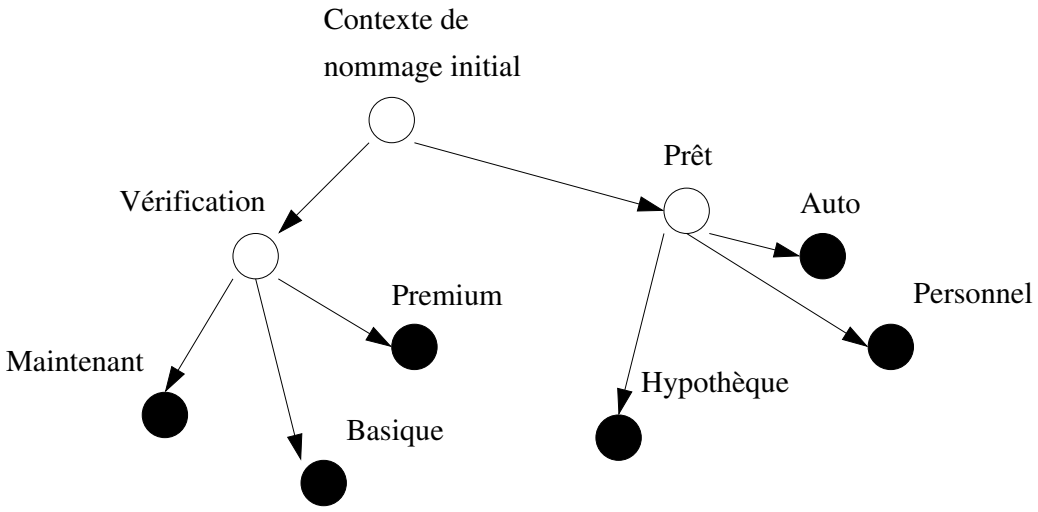

FIG. 2.4 – Hiérarchisation des contextes de nommage

Le service de Trading quand à lui permet la localisation d'objets par leur description. Les clients recherchent des objets à l'aide d'une description et de mots-clés. Le service de Trading leur retourne une liste d'objets qui correspondent aux critères du client.

Le service de nommage est fortement lié à l'ORB. Celui-ci se sert de ce service

pour localiser les autres services. Le fournisseur de services doit modifier le démarrage de l'ORB pour que celui-ci prenne en compte un service de nommage particulier. Certains ORB permettent de modifier le comportement de la phase de démarrage (c'est-à-dire les appels à `resolve_initial_references()`) par une option sur la ligne de commande. Pour les autres ORB il faut modifier le code de la plateforme même.

L'utilisateur de services doit ajouter dans le code métier de l'objet qui doit s'enregistrer dans le service de nommage le code suivant :

```
1  // Récupérer le contexte de nommage racine
2  org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
3  NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
4
5  // Récupérer la référence depuis le servant
6  org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
7  Hello href = HelloHelper.narrow(ref);
8
9  // Référencer l'objet par un nom dans le service de nommage
10 String name = "Hello";
11 NameComponent path[] = ncRef.to_name(name);
12 ncRef.rebind(path, href);
```

L'utilisateur doit modifier son code pour récupérer une référence sur un objet dont il connaît le nom :

```
1  // Récupérer le contexte de nommage racine
2  org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
3  NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
4
5  // Résoudre la référence de l'objet à partir de son nom
6  String name = "Hello";
7  helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
```

2.2.3 Le service de persistance

Le service de persistance permet à l'état d'un objet de survivre après la terminaison du programme qui l'a fabriqué [86]. Un service de persistance fournit deux opérations, le stockage d'un objet et sa récupération (cf fig. 2.5). Trois propriétés des objets rendent ces opérations délicates [105].

D'abord le problème de l'identité d'un objet. Si un objet qui a déjà été stocké est à nouveau stocké, il faut alors mettre à jour la copie déjà stockée. Inversement si un objet qui a déjà été récupéré est récupéré à nouveau il faut alors renvoyer la référence de cet objet.

Ensuite les objets forment un réseau qu'il faut maintenir à travers le stockage et la récupération. Il faut donc pouvoir distinguer les objets essentiels des objets temporaires qui ne seront pas stockés. Le service de persistance doit aussi maintenir les liaisons introduites par le service de relations [82].

Enfin les objets sont des unités d'encapsulation. Le contenu des objets doit être protégé contre des manipulations directes passant à travers l'encapsulation. Il faut

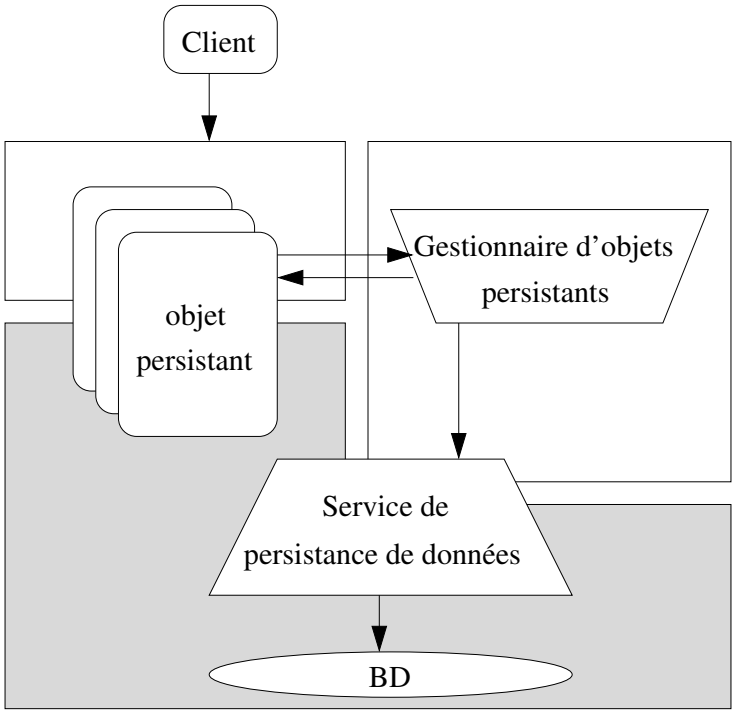

FIG. 2.5 – Fonctionnement du service de persistance

donc passer par des appels de méthodes auxquelles doivent répondre les objets persistants.

Le service de persistance résout ces problèmes par le biais de protocoles entre les objets persistants et le système de stockage. Comme ces protocoles demandent la coopération des objets persistants avec le service de persistance ils sont parfois nommés protocoles conspirants [88].

Plusieurs implantations de la persistance sont possibles [51]. Dans un premier cas l'utilisateur de services doit fournir des méthodes pour stocker (méthode `save`) et récupérer un objet (méthode `restore`), le travail du fournisseur de service est alors de fournir les composants qui permettent de faire la liaison entre l'objet persistant et un support persistant (qui peut être une base de données). Dans un deuxième cas, elle est implémentée dans l'objet d'adaptation (le POA) par le fournisseur de services qui fournit un POA pour la persistance. L'utilisateur n'a alors qu'à instancier ce POA au moment de l'exécution de son code. Il écrit alors par exemple :

```
1  // Récupérer le POA racine
2  POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
3  rootPOA.the_POAManager().activate();
4
5  // Créer un POA avec une politique de persistance
6  Policy[] policy = new Policy[1];
7  policy[0]=rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
8  POA poa = rootPOA.create_POA("PersistentPOA", null, policy);
9  poa.the_POAManager().activate();
10
11 // Créer un servant et lui attribuer une référence pour la persistance
```

```
12 byte[] objectId = "MyPersistentObjectId".toBytes();
13 HelloServant helloServant = new HelloServant();
14 poa.activate_object_with_id(objectId, helloServant);
15 Hello helloRef = helloHelper.narrow(poa.id_to_reference(objectId));
```

Notons qu'une implantation de POA n'est valable que pour un ORB donné. Il faut donc que le fournisseur de service implémente le POA persistant pour les différents ORB dans lesquels il veut intégrer son service.

2.2.4 Le service de sécurité

Le service de sécurité offre un ensemble de services tels que l'authentification, les communications sécurisées, la délégation de droits, la non répudiation [74].

L'identification et l'authentification de rôles se fait par le biais d'une identité (qui peut se vérifier par mot de passe par exemple). L'autorisation et le contrôle d'accès se basent sur les droits et privilèges accordés aux rôles.

Une fois que les objets se sont authentifiés, les communications sécurisées protègent les données qui transitent. Le système de communications sécurisées doit supporter différents protocoles d'authentification (Kerberos, RSA...), différents types d'algorithmes (signature par cheksum, DES...) et doit être extensible.

Dans un système distribué, quand un client effectue un appel de méthode sur un objet, l'objet peut faire appel à d'autres objets pour effectuer une partie du travail. Des décisions de contrôle d'accès doivent être prises à chaque point de la chaîne. On peut trouver plusieurs types de délégations de privilèges. La non délégation permet uniquement à l'objet intermédiaire d'utiliser ses propres droits pour invoquer des opérations déléguées (cf fig. 2.6). La délégation simple permet à l'objet intermédiaire de n'utiliser que les droits du client pour invoquer les opérations déléguées (cf fig. 2.7) tandis que la délégation composée permet à l'objet intermédiaire d'utiliser ses droits et ceux du client pour invoquer les opérations déléguées (cf fig. 2.8).

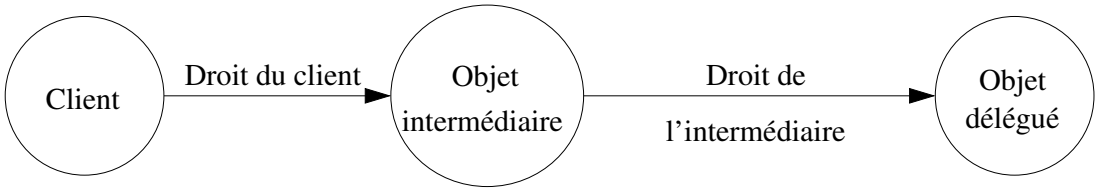

FIG. 2.6 – Service de sécurité : non délégation

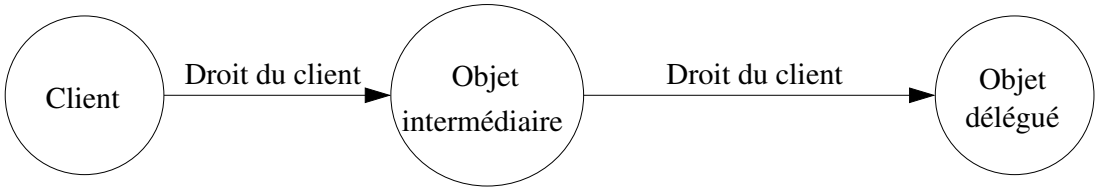

FIG. 2.7 – Service de sécurité : délégation simple

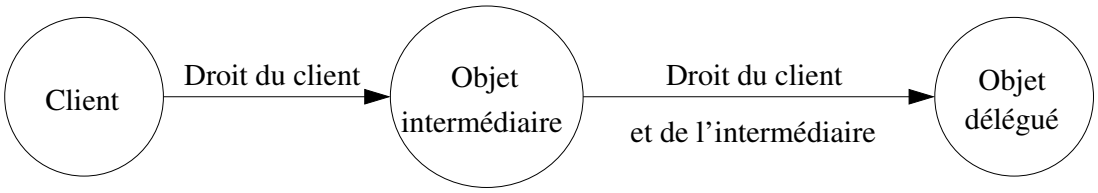

FIG. 2.8 – Service de sécurité : délégation composée

Pour mettre en œuvre l'authentification, l'utilisateur de services doit ajouter dans son code client le code suivant qui permet de s'authentifier par un mot de passe auprès d'un serveur et de préciser que les appels de méthodes utiliseront cette authentification. L'utilisateur du service devra aussi prévoir la liste des utilisateurs et leurs droits. Le code du serveur doit implémenter une interface standardisée pour supporter l'authentification.

```
1  // Création d'un nouveau contexte de sécurité pour
2  // stocker les données d'authentification
3  LoginHelper loginHelper = new LoginHelper(); try {

5  // Indication de l'ID et du mot de passe de l'utilisateur pour
6  // authentification.
7  org.omg.SecurityLevel2.Credentials
8          credentials = loginHelper.login(userid, password);

10 // Utilisation des nouvelles données d'identification
11 // pour tous les appels à venir.
12 loginHelper.setInvocationCredentials(credentials);
```

Pour le fournisseur de services, la tâche est plus complexe. Il doit faire en sorte que les requêtes qui circulent dans l'ORB, entre un client qui s'est authentifié et un serveur, portent les données d'authentification. Il faut donc modifier la plateforme en profondeur pour qu'au moment de l'empaquetage et du dépaquetage des données l'ORB ajoute les données d'authentification (et les crypte éventuellement) pour que ces données puissent être vérifiées du côté serveur. Bien entendu les modifications apportées à un ORB ne sont pas facilement transposables à un autre ORB à cause de l'hétérogénéité des infrastructures.

2.2.5 Le service de transaction

Le service de transaction (OTS) est un des services les plus important pour construire des systèmes distribués fiables [81]. La fiabilité d'une transaction est obtenue par les propriétés ACID : atomicité, cohérence, isolation, durabilité [5].

- **Atomicité :** soit la transaction réussit, soit elle échoue. Il n'y a pas d'état intermédiaire possible.
- **Cohérence :** les invariants définis par le système ne sont pas remis en cause. La source de données ne peut pas être dans un état incohérent durant l'exécution de la transaction.

- **Isolation :** les états intermédiaires ne sont pas visibles depuis une autre transaction.
- **Durabilité :** les modifications validées sont persistantes et ne peuvent être annulées.

L'invocation d'une opération pendant une transaction doit se faire soit sur un objet *transactionnel* soit sur un objet *recouvrable*. Un objet recouvrable est un objet dont les données sont affectées par la réussite ou l'échec de la transaction. Les objets recouvrables contiennent les données et fournissent un mécanisme de récupération d'erreurs. Les objets recouvrables doivent être préparés pour réagir aux messages du gestionnaire de transactions correspondant à un `commit` ou `rollback`. Un objet transactionnel est un objet dont le comportement est affecté par le fait de faire partie d'une transaction. C'est donc soit un objet recouvrable soit un objet qui fait référence à un objet recouvrable. S'il n'est pas un objet recouvrable lui même, il participe à la transaction en passant le contexte transactionnel. Au bout de la chaîne les objets recouvrables utilisent le contexte avec le gestionnaire de transactions pour coordonner toutes les différentes parties de la transaction pour le `commit` ou le `rollback`.

Une transaction est démarrée et terminée par un client (c'est la démarcation). Entre le point de départ et la fin, les opérations sont effectuées sous le contrôle du gestionnaire de transactions et acquièrent les propriétés ACID. Quand le client demande la fin de la transaction, le gestionnaire de transactions demandent à tous les objets participants s'ils sont prêts à terminer la transaction. S'ils répondent tous oui, le gestionnaire de transactions envoie un message de `commit` à tous les participants, sinon il envoie un message de `rollback`. C'est le modèle du `commit` en deux temps.

L'OTS maintient automatiquement le contexte de la transaction courante qui est propagé à tous les participants de la transaction. Les opérations sur les transactions (`begin`, `commit`, `rollback`) sont définies sur le contexte courant.

Tous les objets qui sont modifiés par une transaction et qui requièrent un contrôle transactionnel s'enregistrent auprès de l'objet de coordination des transactions. Le coordinateur en charge de la transaction peut être retrouvé depuis le contexte courant. Une ressource peut indiquer si le coordinateur supporte les transactions imbriquées. Les ressources sont utilisées par le coordinateur pour exécuter le protocole du `commit` en deux temps.

Une implantation de l'OTS doit supporter les transactions plates et éventuellement les transactions imbriquées. Les transactions imbriquées permettent la création de nouvelles transactions à l'intérieur d'une transaction existante. La transaction existante est appelée *parente* et la transaction imbriquée est une sous transaction que l'on nomme *fille*. Une transaction ne peut pas émettre un `commit` tant que tous ses enfants n'ont pas `commité`. Quand une transaction est en `rollback` tous ses enfants sont mis en `rollback`.

On résume le service de transaction sur la figure 2.9.

L'OMG requiert un certain nombre de caractéristiques d'un système de transaction tel que le support pour différents modèles de transaction, un déploiement évolutif, l'interopérabilité des modèles, l'interopérabilité du système (un service de transaction sur un ORB, plusieurs services de transactions sur un ORB, un service de transactions sur plusieurs ORB, plusieurs services de transactions sur plusieurs ORB), le support des transactions non partagées, un contrôle flexible de la propagation des transactions, de la synchronisation, le support de moniteurs de transactions et le support des standards existants de transactions. Détaillons l'intégration de ce

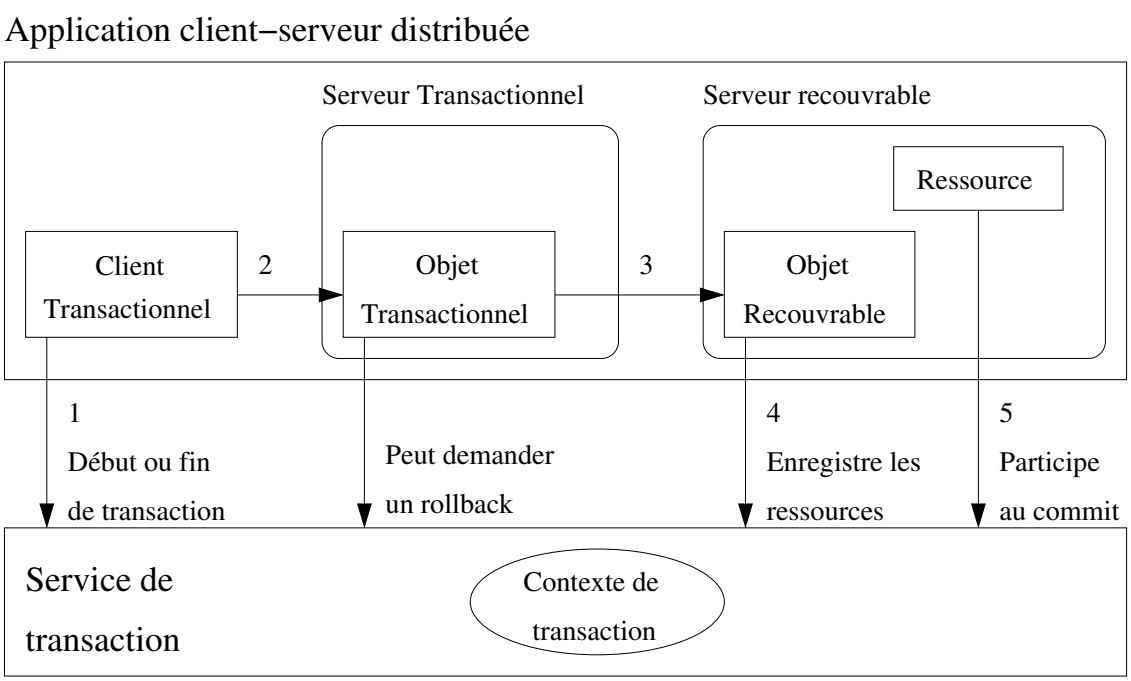

FIG. 2.9 – Fonctionnement du service de transactions

service.

L'utilisateur du service de transactions doit d'abord du côté client initialiser le service dans l'ORB. Cela se fait en demandant une référence à l'ORB (ligne 2) puis en typant correctement cette référence à l'aide de l'opération `narrow` (ligne 3).

```
1   ORB orb = ORB.init(args, null);

3   Object obj = orb.resolve_initial_references("TransactionCurrent");
4   Current tx_current = CurrentHelper.narrow(obj);
```

Ensuite il doit créer une nouvelle transaction (ligne 1). Il envoie les messages qui doivent être effectués pendant la transaction (ligne 3). Puis il tente de terminer la transaction en demandant au contexte de *commiter* (ligne 6). Le client de la transaction peut aussi demander le `rollback` de la transaction en appelant la méthode `tx_current.rollback()` à la place de la méthode `commit`.

```
1  tx_current.begin();

3  // Code à exécuter dans la transaction

5  try {
6    tx_current.commit(true)
7  } catch (TRANSACTION_ROLLEDBACK) {
8    // La transaction est en rollback
9  }
```

Du côté des objets qui participent à la transaction, pour pouvoir utiliser le contexte, ils doivent utiliser un POA transactionnel. Tout d'abord l'utilisateur du service doit définir la police d'utilisation des transactions (ligne 4 à 8). Ensuite il doit récupérer un POA racine (ligne 10 et 11) pour pouvoir créer un POA qui gère le passage de contexte pour la transaction (ligne 13 et 14). Enfin il crée l'objet métier qui participe à la transaction en l'activant sur le POA transactionnel (ligne 16 à 19).

```
1  // Initialisation de l'ORB.
2  ORB orb = ...

4  // Creation d'un objet de propriété pour la propriété REQUIRES OTS.
5  Any policy_val = orb.create_any();
6  OTSPolicyValueHelper.insert(policy_val, REQUIRES.value); Policy
7  tx_policy = orb.create_policy(OTS_POLICY_TYPE.value,policy_val);
8  Policy[] policies = new Policy[1]; policies[0] = tx_policy;

10 // On récupère une référence sur le POA.
11 Object obj = orb.resolve_initial_references("RootPOA"); POA
12 root_poa = POAHelper.narrow(obj);

14 // On crée un nouveau POA avec la propriété OTS.
15 POA tx_poa =
16     root_poa.create_POA("REQUIRES TX",root_poa.the_POAManager(),policies);

18 // Finalement on crée l'objet métier en se servant du POA transactionnel.
19 AccountImpl servant = new AccountImpl(...);
20 byte[] id = tx_poa.activate_object(servant);
21 obj = tx_poa.servant_to_reference(servant);
22 Account account = AccountHelper.narrow(obj);
```

Pour le fournisseur de service, en plus de fournir un POA transactionnel, il doit comme pour le service de sécurité passer en plus dans les requêtes le contexte transactionnel créé par le client de la transaction de manière transparente. Comme pour le service de sécurité les modifications apportées à l'ORB ne sont pas facilement transposables à d'autres ORB.

2.3 Analyse de l'utilisation des services dans CORBA

Les exemples précédents nous donnent un aperçu du rôle du fournisseur de services et de celui de l'utilisateur de services. Pour pouvoir intégrer un service dans son application, l'utilisateur de service doit recevoir de la part du fournisseur de services les données suivantes :

- un ensemble d'interfaces du service qui correspondent aux appels que peut envoyer l'utilisateur au service.
- des objets de service qui implémentent ces interfaces.
- des interfaces que doivent implémenter les objets de l'application pour pouvoir répondre aux appels du service.

– de la documentation qui explique les interfaces à implémenter, les classes à sous-classer, les messages à envoyer aux objets de services avec quel format et quels paramètres.

La répartition du travail d'intégration de services entre le fournisseur de services et l'utilisateur de services varie.

Soit l'utilisateur doit faire tout le travail d'appel et d'utilisation du service et donc d'extension de la plateforme. Dans ce cas le travail d'intégration du fournisseur est quasi nul puisqu'il ne fait que fournir les objets de services, les interfaces d'appel et la documentation. L'utilisateur doit modifier son application en conséquence, c'est à dire implémenter les interfaces de rappels, modifier l'héritage, ajouter des appels d'initialisation des objets de services, insérer les appels vers le service, modifier les signatures des méthodes. Quand l'utilisateur veut utiliser plusieurs services il doit en plus de faire l'intégration de chaque service dans son application et penser à la façon dont il entremêle les appels vers les différents services pour garder une cohérence d'appel entre les services (on parle de composition de services). Toute la complexité de l'intégration repose sur l'utilisateur. Par contre il en découle une grande adaptabilité de l'intégration de services par l'utilisateur puisqu'il a tout le contrôle. Guerraoui dans [50] démontre la nécessité d'une telle souplesse.

Soit le fournisseur de services intègre une partie de la gestion du service dans l'ORB, ce qui simplifie la tâche de l'utilisateur. Si l'on prend comme exemple le service de transactions, l'utilisateur n'a pas à s'occuper du passage du contexte entre les objets, c'est fait automatiquement par la plateforme. Dans ce cas le travail du fournisseur est complexe. Il doit identifier les objets qui définissent le fonctionnement de l'ORB pour pouvoir y insérer les points d'extension qui vont permettre une utilisation plus simple. Par exemple le passage d'un objet de contexte dans le cas des transactions entre les différents objets qui sont sous le contrôle de la transaction se fait en ajoutant le contexte dans les entêtes `ServiceContext` des requêtes et réponses GIOP (General Inter-ORB Protocol). Les modifications de l'ORB pour intégrer le service de transactions sont profondes puisque les objets de toutes les couches de l'ORB sont impactés. Étant donné que les implantations des ORB sont différentes le portage d'un service est complexe. Le problème de l'hétérogénéité des ORB empêche la diffusion des implantations de services non standardisées ou qui exigent des modifications profondes de l'infrastructure pour être diffusé. De plus quand il y a plusieurs services à intégrer au cœur de l'ORB, le fournisseur doit faire attention à la composition des services dans les objets de fonctionnement de l'ORB. Par exemple il faut un POA pour faire de la sécurité, il en faut un autre pour faire des transactions et il en faut un troisième pour avoir accès aux services de transactions et de sécurité en même temps. Il y a une multiplication du coût à l'ajout de nouveaux services. Et il faut fixer les règles de composition.

Pourtant nous voyons que le protocole pour intégrer un service donné est toujours le même. Pour l'utilisateur il y a une répétitivité des services à démarrer, des appels à effectuer, des interfaces à implémenter et des méthodes à écrire. Le fournisseur de services de son côté modifie des points clés de la plateforme qui sont toujours les mêmes d'un point de vue conceptuel, mais les liens entre les services sont différents.

Conclusion

Dans ce chapitre nous avons étudié les services standard ainsi que leur intégration dans l'intergiciel de communication CORBA. Nous avons évalué la répartition du travail d'intégration entre le fournisseur de services et l'utilisateur de services. Nous avons vu que le manque d'automatisation de l'intégration de services engendre la répétitivité d'une tâche assez complexe.

La table 2.3 résume les propriétés d'intégration de services dans CORBA 2.0. Nous rappelons que dans notre vocabulaire le fournisseur de services est celui qui écrit les services et que l'utilisateur de services est celui qui se sert des services dans ses applications.

Critère	Évaluation
Gestion de l'hétérogénéité / Réutilisation	**bonne** si le service n'est pas intégré dans l'ORB, **mauvaise** si le service est intégré dans l'ORB
Séparation des intégration de services	**bonne** *mais composition manuelle* pour l'utilisateur (et pour le fournisseur si le service est intégré dans l'ORB)
Adaptabilité de l'intégration	**bonne** puisque l'utilisateur doit intégrer le service dans son application et donc peut l'adapter à son besoin *mais engendre de la complexité*
Découplage du travail du fournisseur et de l'utilisateur	**mauvais** puisque uniquement par la documentation et l'API
Complexité de mise en œuvre	**forte** pour l'utilisateur et **forte** pour le fournisseur s'il fait l'intégration dans l'ORB

TAB. 2.3 – Résumé des propriétés d'intégration de services en CORBA 2.0

Dans le chapitre suivant nous nous intéressons aux plateformes à composants qui automatisent l'intégration de services dans les composants. Nous étudions les mécanismes qui permettent cette automatisation et nous nous interrogeons sur la facilité d'ajouter de nouveaux services dans ces plateformes à composants.

CHAPITRE 3

Intégration de services dans les plateformes à composants

Dans ce chapitre nous étudions l'intégration de services dans les plateformes à composants. Les plateformes à composants sont une évolution des intergiciels de communication. Les composants avec leur plateforme d'exécution permettent une meilleure séparation du code métier et du code technique (que sont les services de notification, de nommage, de persistance, de sécurité, de transaction, etc.). La gestion de la configuration, du déploiement et de l'exécution est aussi facilitée dans ces plateformes.

De manière générale le programmeur de composants fournit la partie métier et le système se charge de générer la partie technique adéquate qui comprend le code d'appels aux services. La structure des composants fait coopérer ces deux parties de code en suivant un modèle dans lequel le code technique encapsule le code métier. Le code de gestion technique est appelé *conteneur* et gère les services. Les conteneurs ont deux rôles principaux : ils gèrent le cycle de vie (création, accès, activation, passivation, destruction, etc.) des composants qu'ils contiennent et leur fournissent automatiquement et de manière transparente les services techniques par interception. Le paramétrage et l'utilisation de ces services techniques sont effectués de manière déclarative, au moment du déploiement, au travers de descripteurs de déploiement.

Les plateformes à composants facilitent la tâche du développeur d'applications qui utilise des services. Son travail consiste à fournir un descripteur de déploiement avec l'ensemble des services qu'il veut utiliser et les paramètres à passer à ces services. Ce paramétrage par le descripteur de déploiement contraint le degré d'adaptabilité du service par l'utilisateur. D'un coté si l'utilisateur a trop peu de paramètres sur lesquels jouer il est contraint et d'un autre coté s'il a trop de paramètres à gérer il risque de se perdre dans la configuration.

Dans ce chapitre nous étudions trois modèles de composants que sont les EJB, les CCM et Fractal. Pour le modèle des EJB [103] nous étudions deux implantations : JOnAS [69] et JBoss [45]. Pour les CCM [84] nous étudions l'implantation OpenCCM [70] et pour Fractal [27] l'implantation Julia [68]. Nous cherchons à mettre en avant la difficulté pour le fournisseur de services d'intégrer de nouveaux services dans ces plateformes à composants, ainsi que de porter l'intégration d'un service d'une plateforme à l'autre. Notons quand même que l'extensibilité n'est pas un des objectifs premiers des EJB.

Cette étude des modèles à composants et de leurs implantations a deux objectifs. Premièrement elle nous permet de mieux cerner le rôle des différents objets dans les plateformes à composants qui font l'hétérogénéité de ces infrastructures. Deuxièmement elle nous permet d'évaluer le paramétrage des services offerts par les outils qui automatisent l'intégration de services dans ces plateformes et nous montrent la difficulté d'ajouter un nouveau service dans ces plateformes à composants.

3.1 Enterprise JavaBeans

Nous décrivons tout d'abord la spécification EJB telle que définie par Sun. Puis nous étudions deux implantations différentes de cette spécification que sont JOnAS et JBoss. Nous mettons en avant la différence des structures de ces deux implantations de la même spécification.

3.1.1 Le modèle EJB

La spécification EJB a été définie par Sun dans le cadre de J2EE (Java 2 Enterprise Edition). Le but principal de la spécification EJB est de faciliter et normaliser le développement, l'assemblage et le déploiement des composants EJB sur des plateformes conformes à la spécification. La spécification décrit l'architecture de l'environnement pour l'exécution des EJB : la définition du serveur EJB et du conteneur EJB ainsi que des services fournis par l'environnement d'exécution aux composants EJB. Les principaux services sont : les transactions, le support pour la désignation et la distribution des objets, le support pour la persistance et la sécurité. Ces services se présentent aux composants EJB comme des ressources fournies par le serveur EJB. Le conteneur est la matérialisation d'une couche architecturale qui s'interpose entre chaque composant et le serveur afin d'assurer l'indépendance du composant vis-à-vis de l'implantation particulière du serveur. Il s'interpose aussi entre le client et le composant, ce qui permet l'accès transparent aux services. En cela, il joue un rôle similaire au POA étudié précédemment.

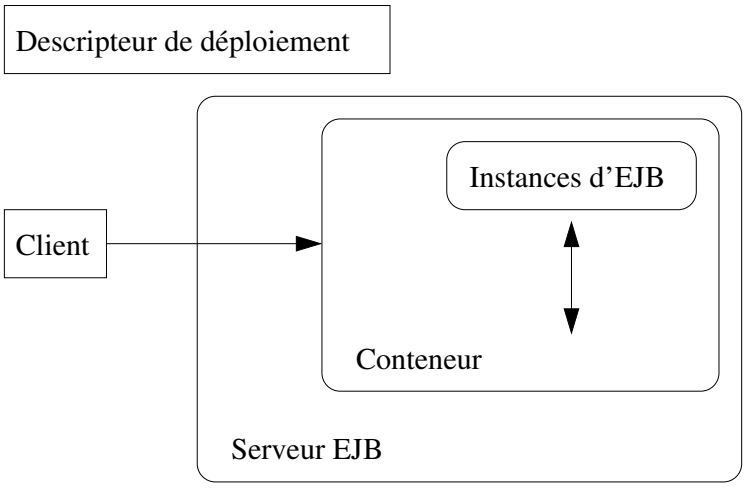

FIG. 3.1 – Serveur et conteneur de la spécification EJB

La programmation des composants EJB est indépendante du système, et même

de la plateforme EJB sur laquelle le composant est déployé. Un composant EJB comporte : une interface définissant les fonctions du composant, une interface contenant les opérations de création, suppression et recherche, une classe qui réalise les fonctions du composant ainsi que des méthodes spécifiques imposées par la spécification EJB et qui sont nécessaires à la gestion du cycle de vie du composant au sein d'un conteneur EJB. Un descripteur de déploiement décrit le composant et définit ses besoins en terme de ressources nécessaires à son exécution.

Un conteneur est constitué, au minimum, d'une usine à composant et d'objets d'interposition. L'usine à composants, ou EJB Home, permet de créer de nouveaux composants à l'intérieur du conteneur, de retrouver des composants existants, et de détruire des composants. Il y a une usine à composants par conteneur et par type de composant. Les objets d'interposition, ou EJB Objects, permettent d'accéder aux composants encapsulés à l'intérieur du conteneur. En effet, ces composants ne sont jamais accédés directement depuis l'extérieur, mais toujours en passant par un objet d'interposition (d'où son nom). Il y a un objet d'interposition par composant encapsulé (sauf éventuellement pour les composants dits sans état, ou *stateless bean*).

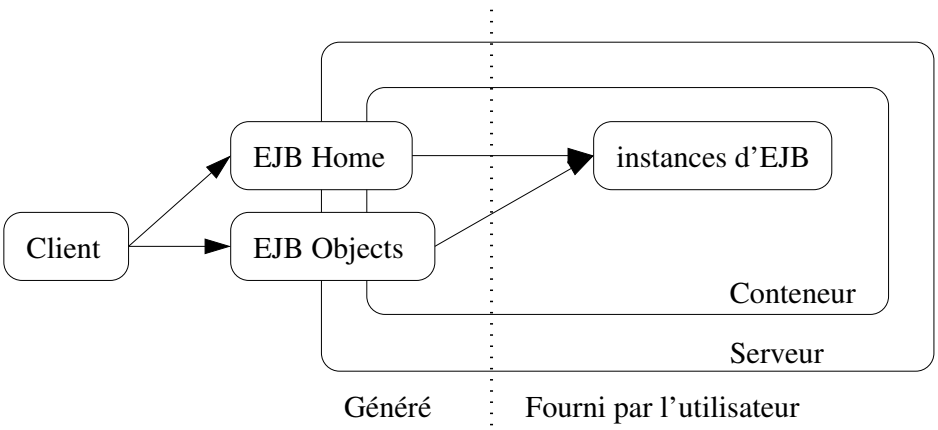

FIG. 3.2 – EJB Home et EJB Objects de la spécification EJB

Un objet d'interposition a avant tout, un rôle équivalent à celui d'un squelette dans CORBA : il dépaquette les messages d'invocation à distance reçus sur le réseau, invoque la méthode correspondante du composant encapsulé, empaquette le résultat de cet appel, puis le retourne dans la réponse. De plus un objet d'interposition peut effectuer certaines opérations juste avant et juste après avoir invoqué la méthode sur le composant à la manière des intercepteurs introduit dans la nouvelle proposition CORBA 3 (cf. section 3.2.1). Le modèle EJB prévoit que le code des objets d'interposition, ainsi que celui des usines à composants, soit généré automatiquement à partir d'un fichier de déploiement. Ce fichier de déploiement est écrit par l'utilisateur en fonction de ses besoins, avant de déployer son application.

Comme nous allons le voir dans les deux exemples d'implantation, les plateformes EJB actuelles utilisent des objets d'interposition monolithiques qui limitent l'extensibilité des plateformes EJB. Pour ajouter un nouveau service, il faut réécrire l'objet d'interposition. Comme celui-ci est généré par un générateur, il faut modifier le code de ce générateur, ce qui demande en plus de comprendre l'architecture du générateur, une bonne compréhension de la combinaison des services existants pour les combiner avec le nouveau service à intégrer.

3.1.2 JOnAS

JOnAS est une implantation de la spécification EJB réalisé par Bull [69]. JOnAS fournit l'outil GenIC qui se charge de la génération des classes d'interposition, qui constituent le conteneur EJB, dans lesquelles les informations fournies dans le descripteur de déploiement sont concrétisées. L'architecture de l'implantation de JOnAS (version 2.5) pour un bean entité (qui est destiné à un usage métier) est composé pour la partie serveur d'un squelette RMI, d'un objet d'interposition (qui correspond à l'EJB Object de la spécification et nommé l'objet Remote dans JOnAS), des services qui sont attachés à l'objet d'interposition et le bean. La figure 3.3 montre ces objets et leurs liaisons.

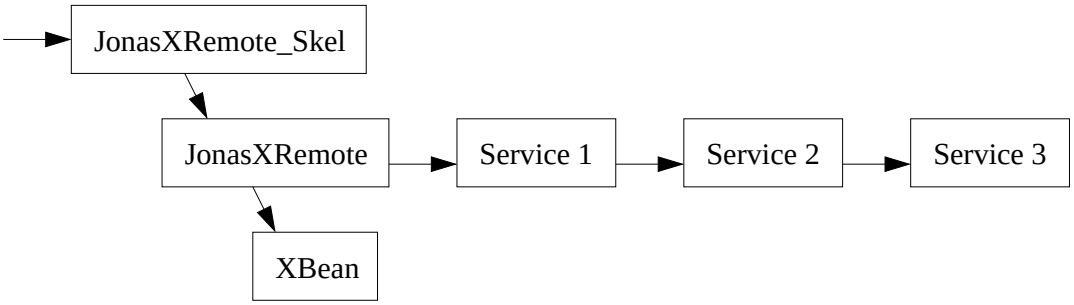

FIG. 3.3 – Architecture de l'implantation JOnAS

Le client d'un composant récupère un stub qui communique par RMI avec le squelette de l'objet d'interposition. Si nous analysons le code de l'objet d'interposition nous pouvons voir que chaque corps de méthode ressemble au code suivant :

```
1  public double getBalance() throws java.rmi.RemoteException {
2      TraceEjb.interp.log(BasicLevel.DEBUG, "");
3      double result;
4      String secu = "";
5      RequestCtx rctx = preInvoke(2, secu);
6      try {
7          // bs est un JEntitySwitch hérité de JEntityRemote
8          JEntityContext bctx = bs.getICtx(rctx.currTx);
9          eb.JOnASAccountImplBean b =
10                 (eb.JOnASAccountImplBean) bctx.getInstance();
11         result = b.getBalance();
12     } catch (RuntimeException e) {
13         rctx.sysExc = e;
14         throw new RemoteException("RuntimeException thrown by
15                                    an enterprise Bean", e);
16     } catch (Error e) {
17         rctx.sysExc = e;
18         throw new RemoteException("Error thrown by an enterprise Bean", e);
19     } catch (RemoteException e) {
20         rctx.sysExc = e;
21         throw e;
22     } finally {
```

```
23          postInvoke(rctx);
24      }
25      return result;
26  }
```

C'est dans ce code et dans les appels `preInvoke` (ligne 5) et `postInvoke` (ligne 22) que sont localisés les appels vers les services. Les objets d'interposition sont générés par GenIC. Pour intégrer de nouveaux services dans JOnAS le fournisseur de services doit modifier le générateur GenIC. S'il veut également faciliter la tâche de l'utilisateur, il devra également modifier le parser du descripteur de déploiement pour pouvoir récupérer les paramètres passés par l'utilisateur de services au nouveau service.

La modification du générateur GenIC est discutée dans [3] avec comme exemple l'intégration d'un service de communications asynchrones avec futurs. Les principales difficultés sont la composition avec les services déjà présents et la complexité de la modification du générateur. Le fournisseur de services doit connaître les détails de l'implantation du générateur pour pouvoir le modifier et ajouter les appels vers le nouveau service.

3.1.3 JBoss

JBoss est une autre implantation de la spécification EJB. JBoss est basé sur une architecture logicielle différente de celle de JOnAS comme le montre la figure 3.4.

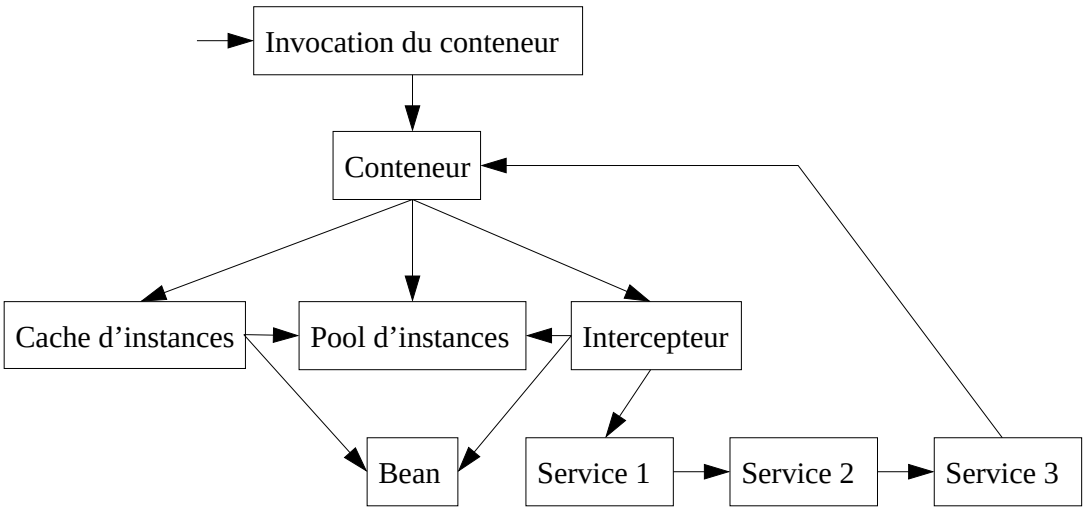

FIG. 3.4 – Architecture de l'implantation JBoss

L'EJB Object de la spécification est représenté dans JBoss par un ensemble d'objets que sont l'invocation du conteneur, le conteneur, l'intercepteur, le cache et le pool d'instances. Le code de gestion des services est fragmenté dans ces différents objets.

La difficulté pour le fournisseur de services est identique à celle expérimentée dans JOnAS pour l'intégration de nouveaux services. Il est intéressant de constater que pour une même spécification, les implantations varient fortement. C'est une difficulté supplémentaire pour le fournisseur de services qui doit faire face pour une même spécification à l'hétérogénéité des implantations.

3.2 CORBA Component Model

Nous étudions maintenant la spécification CCM en la comparant à la spécification EJB et nous présentons l'implantation OpenCCM.

3.2.1 Le modèle CCM

Spécifié par l'Object Management Group en 2002, le CCM est basé sur CORBA qui permet l'utilisation d'environnements d'exécution hétérogènes.

Les composants CCM sont définis par un ensemble d'attributs et par leurs interfaces fournies et requises, appelés respectivement ports d'entrée et ports de sortie. Les attributs de composants sont utilisés pour la configuration au déploiement ou à l'exécution. Les ports définissent les points de communication des composants et sont utilisés pour leur invocation et leur interconnexion. En effet, il n'est pas possible d'appeler un composant sans référencer un de ses ports d'entrée. Pour que les composants d'une application puissent communiquer, ils doivent être interconnectés à l'aide de leurs ports. Les connexions sont établies entre des ports de sortie et des ports d'entrée qui représentent des interfaces identiques et qui sont de la même nature synchrone ou asynchrone. L'acheminement des communications est pris en charge par le bus CORBA. CCM définit les types de composants, ainsi que leurs implantations, à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL de CORBA qui permet de programmer les composants dans des langages différents et de les exécuter dans des environnements hétérogènes. Les implantations des composants sont décrites à l'aide du langage de description CIDL. Les descriptions CIDL sont utilisées pour la génération des conteneurs. Elles contiennent des détails sur la structure logicielle et sur la gestion des services techniques du composant. Comme dans les EJB, ces conteneurs s'exécutent dans des serveurs qui leur fournissent les services nécessaires (cf. figure 3.5).

Contrairement aux conteneurs EJB, un conteneur CCM ne gère les instances que d'un type de composant. Les conteneurs contiennent des usines de composants (de type *Home*) qui se chargent de la création et de la destruction des instances du composant. Ils disposent également d'objets d'interposition pour les interfaces fournies et requises des instances du composant. Les conteneurs proposent deux interfaces particulières de gestion des objets d'interposition : une d'introspection et une de gestion de ports. L'interface d'introspection fournit des informations sur le type de composant, sur ses interfaces fournies et requises, ainsi que sur l'état de ses connexions à d'autres composants. L'interface de gestion de ports est utilisée pour établir ou détruire des connexions entre composants. En effet, les interconnexions entre composants sont gérées à l'aide de primitives explicites et ne sont pas établies par simple passage de référence comme dans le cas des EJB. Les interfaces d'introspection et de gestion de ports, font partie des interfaces externes des conteneurs qui incluent également les interfaces fonctionnelles des composants, ainsi que l'interface de l'objet *Home*.

Le contrôle au déploiement et à l'exécution CCM suit l'exemple des EJB et fournit des descripteurs de déploiement. Toutefois, ces descripteurs concernent non seulement les composants isolés, mais également les assemblages de composants (non décrits par les EJB) pour former des applications. En ce qui concerne les descripteurs de composants, ils décrivent leur structure logicielle, leur type en termes d'interfaces fournies et requises, ainsi que leur catégorie (processus, entité, session, service). Ces

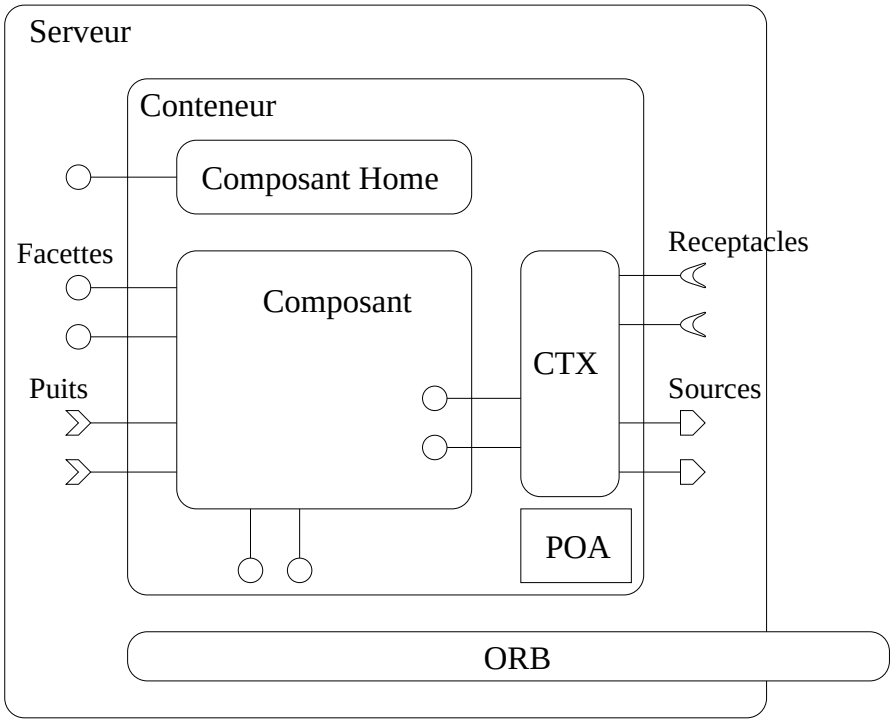

FIG. 3.5 – Serveur et conteneur de la spécification CCM

descripteurs sont générés à la compilation de la description des composants, mais peuvent être modifiés pour paramétrer la gestion des services. Quant aux descripteurs d'assemblages, ils décrivent les architectures initiales des applications. Ils spécifient les instances de composants constituant une application, définissent des règles de placement sur des sites d'exécution et donnent les connexions à établir entre instances.

3.2.2 OpenCCM

OpenCCM [70] est une implantation de la spécification CCM. L'architecture d'OpenCCM telle que décrite dans [107] est représentée figure 3.6.

Un composant de contrôle fournit une abstraction d'une fonction système à la couche de coordination au travers d'une interface générique. Chaque composant de contrôle encapsule le code d'appel du service technique qu'il représente. Le coordinateur définit un modèle de composition pour les contrôles et une abstraction d'un ensemble cohérent de fonctions à la couche d'interception au travers d'une interface générique. L'intercepteur rajoute du comportement lors de l'appel d'opérations sur le composant en effectuant des indirections vers le coordinateur.

L'implantation OpenCCM est moins monolithique dans son approche que JOnAS, mais la complexité de la modification du générateur reste grande et il faut aussi modifier la gestion des descripteurs de déploiement. Le fournisseur de services fait face aux deux difficultés que sont la complexité de la modification du générateur et l'hétérogénéité des différentes implantations.

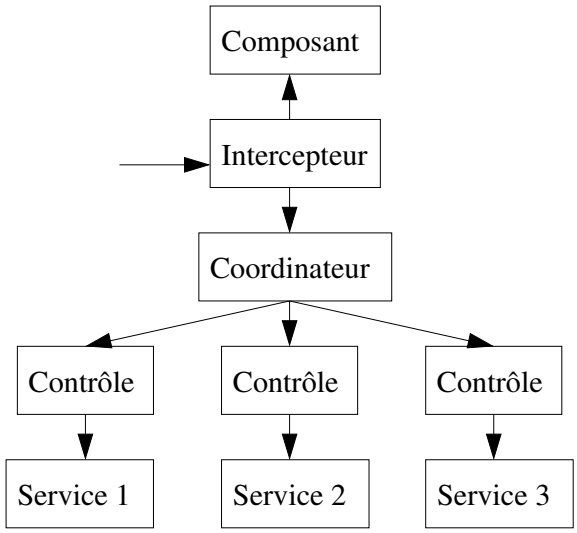

FIG. 3.6 – Architecture de l'implantation OpenCCM

3.3 Fractal

3.3.1 Le modèle Fractal

Le modèle Fractal est un système général, minimal et extensible de relations entre des concepts. Fractal n'est pas dédié à un langage ou un environnement d'exécution particulier. C'est un modèle qui ne présuppose pas une sémantique ou une granularité particulière associée aux composants - à l'inverse des modèles de composants industriels EJB ou CCM qui modélisent des composants « métiers », dont les « conteneurs » fournissent des services techniques.

Le modèle est principalement défini par cinq concepts : composant, interface, liaisons, contrôleur, contenu. Notons que si les concepts d'interface et de liaison sont relativement classiques, Fractal généralise leur usage et induit ainsi une séparation nette entre interfaces et implantations des composants et rend les liaisons manipulables, ce qui favorise les reconfigurations (cf. figure 3.7).

Le contenu d'un composant peut être composé d'autres composants (appelés sous-composants) qui sont sous le contrôle du contrôleur du composant englobant. Le modèle est complètement récursif et autorise l'imbrication des composants à un niveau arbitraire. Un composant peut interagir avec son environnement par l'échange de signaux. Un signal consiste en un nombre fini d'arguments qui peuvent prendre trois formes : un nom, une valeur ou un composant. Les noms sont des symboles utilisés pour désigner des entités du modèle. Les valeurs sont construites à partir de types primitifs et de constructeurs de types de données. Le comportement d'un composant est défini par un ensemble de transitions dans lequel chaque transition spécifie le composant originel, un ensemble fini de signaux entrants, un ensemble fini de signaux sortants et un ensemble fini de composants résultants.

Le contrôleur d'un composant incarne le comportement de contrôle associé à ce composant. Le contrôleur d'un composant peut intercepter les signaux entrants ou sortants de ce composant ou bien superposer un comportement de contrôle. Chaque contrôleur réalise la sémantique de composition de ses sous-composants.

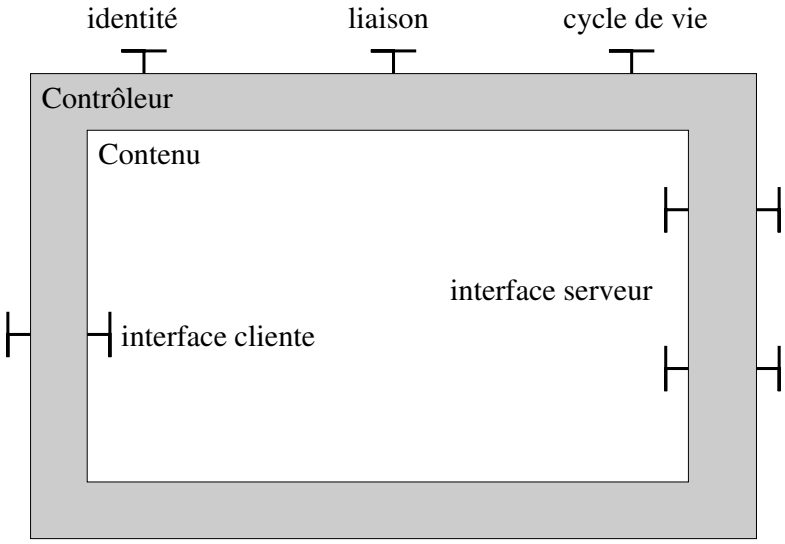

FIG. 3.7 – Eléments de la spécification Fractal

3.3.2 Julia

Julia se présente avant tout comme un framework de programmation et de composition de contrôleurs Fractal en Java. Les contrôleurs Fractal étant des entités abstraites réalisant diverses fonctions de contrôle (liaison, contenu, cycle de vie, nommage...).

Les objets contrôleurs sont de deux types : intercepteurs et objets de contrôle (cf. figure 3.8). Julia fournit un mécanisme pour associer des intercepteurs aux interfaces ; et surtout un mécanisme de mixins [37] pour composer les aspects de contrôle.

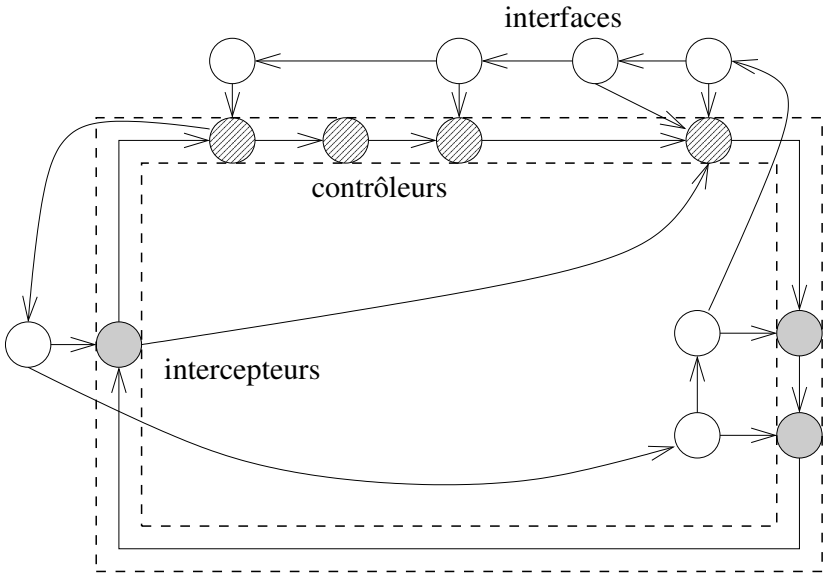

FIG. 3.8 – Architecture de l'implantation Julia

Julia offre deux mécanismes d'optimisation, intra et inter composants. Le premier

mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant les objets de contrôle en un seul. Le second mécanisme permet d'optimiser les chaînes de liaisons entre composants : ce mécanisme permet de traverser les contrôleurs sans intercepteurs, sans pour autant perdre la méta-information sur les liaisons. Julia est constitué d'une bibliothèque d'exécution, d'une bibliothèque de mixins, et de générateurs de code pour générer les interfaces des composants et les intercepteurs, pour mélanger les mixins choisis et en faire des objets de contrôle complets.

Julia facilite le travail du fournisseur de services quant à la gestion des appels vers les services grâce aux différents mécanismes d'interception. Mais c'est toujours au fournisseur de service de faire attention à la composition des services. La configuration des applications se faisant de manière programmatique, le fournisseur de services n'a pas à gérer la lecture du descripteur de déploiement.

Conclusion

Nous avons vu dans ce chapitre différentes spécifications et implantation de plateformes à composants. Le travail de l'utilisateur est facilité par ces approches. Il doit uniquement écrire le descripteur de déploiement dans lequel il précise en plus de la configuration de son application les services qui doivent être appliqués sur les différents composants ainsi que les paramètres à passer à ces services. A l'aide de ces descripteurs de déploiement les générateurs génèrent le code d'interposition qui va contrôler l'exécution des composants. C'est l'avantage majeur des plateformes à composants. Nous avons discuté dans l'introduction de ce chapitre le manque de flexibilité dans l'adaptation des services dus à ce paramétrage par le descripteur de déploiement.

Pour le fournisseur de services la tâche est plus complexe. Dans les plateformes à composants, à l'inverse de l'approche de CORBA 2.0, il doit réaliser entièrement l'intégration des services dans l'infrastructure. Nous avons vu que les différentes plateformes à composants se servent de générateurs pour fabriquer les objets d'interpositions dans lesquels les services sont localisés. Le fournisseur de service doit connaître les détails de l'infrastructure de la plateforme pour pouvoir modifier le générateur. L'hétérogénéité des implantations des plateformes rend le travail d'intégration non portable. De plus il doit être capable de composer les appels vers le nouveau service avec les appels vers les autres services.

La table 3.1 résume les propriétés d'intégration de services dans les plateformes à composants.

Le problème principal pour les approches EJB et CCM est que les générateurs de conteneurs sont fermés et n'offrent pas de possibilité de modification du code généré autre que modifier le code source des générateurs. Nous allons donc chercher du côté des approches de séparation des préoccupations qui permettent de modifier le comportement des applications sans modifier le code source directement.

Le chapitre suivant discute des techniques et outils qui permettent la séparation et la réintroduction de code technique dans le code des applications comme la métaprogrammation, la programmation par tissage et les approches par modélisation. Nous les analysons sur les critères de gestion de l'hétérogénéité, de séparation de services, d'adaptabilité de l'intégration, du découplage fournisseur utilisateur et de la complexité de mise en œuvre.

Critère	Évaluation
Gestion de l'hétérogénéité	**mauvaise** pour le fournisseur de services car les infrastructures divergent
Séparation des services	**mauvaise** car les appels vers services sont définis dans le générateur
Adaptabilité de l'intégration	**moyenne** puisque l'utilisateur n'a comme latitude que les paramètres du descripteur de déploiement
Découplage fournisseur / utilisateur	**bon** grâce au descripteur de déploiement mais problème usuel du choix d'un bon paramétrage
Complexité de mise en œuvre	**forte** pour le fournisseur et **faible** pour l'utilisateur

TAB. 3.1 – Résumé des propriétés d'intégration de services des plateformes à composants

CHAPITRE 4

Outils pour l'intégration de services

Dans ce chapitre nous étudions différentes techniques et outils qui permettent de séparer le code technique du code des applications tels que la métaprogrammation, les approches par tissage et les approches par modélisation.

Nous avons vu que pour l'utilisateur de services les approches par génération des plateformes à composants lui permettent de simplement déclarer dans un descripteur de déploiement les paramètres des services qu'il utilise. La tâche du fournisseur est plus complexe pour offrir à l'utilisateur de nouveaux services dans les plateformes à composants. Pour cela nous étudions ces différentes approches de la séparation des préoccupations qui vont nous permettre d'évaluer quelle technique est adéquate pour faciliter le travail d'intégration de services par le fournisseur de services.

Nous évaluons ces différentes techniques d'intégration de services dans le contexte des plateformes à composants. Les critères d'analyse sont comme précédemment la gestion de l'hétérogénéité, la séparation de services, l'adaptabilité de l'intégration, le découplage entre le fournisseur et l'utilisateur et la complexité de mise en œuvre.

4.1 Métaprogrammation et implantations ouvertes

La métaprogrammation permet d'inspecter et de contrôler un système. Plusieurs techniques de contrôle ainsi que plusieurs niveaux de contrôle sont possibles. Nous examinons une des techniques de métaprogrammation la plus courante qui est l'utilisation de protocoles à métaobjets. Nous étudions aussi la métaprogrammation basée sur des systèmes de description de la communication entre objets qui permettent une meilleure abstraction des éléments du langage.

4.1.1 Protocoles à métaobjets

Un MOP (Protocole à MétaObjets) est composé d'un ensemble de points d'entrées dans le langage dont la spécialisation permet l'introduction de nouveaux comportements [92]. Le contrôle et l'extension des comportements se fait donc sans modifier le code d'implantation de l'application mais en spécialisant les points d'entrées définis par le MOP. Différents types de MOP permettent différents niveaux de spécialisa-

tion. Nous prenons en exemple le MOP de CLOS [9] qui est un des plus complets en terme de points d'entrées.

Les éléments fondamentaux des programmes CLOS (les classes, les variables d'instance, les fonctions génériques, les méthodes, etc.) sont représentés par des objets de première classe (cf. figure 4.1). Le comportement de CLOS est défini par les méthodes de ces objets de première classe. On appelle ces objets des métaobjets parce qu'ils décrivent le comportement de CLOS lui-même. Le MOP de CLOS est formé par le protocole suivi par ces métaobjets pour fournir le comportement de CLOS [47].

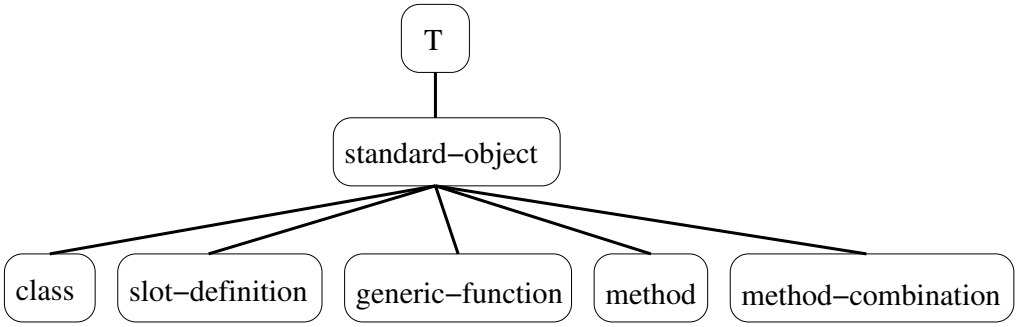

FIG. 4.1 – CLOS : hiérarchie des classes du MOP

Le MOP de CLOS est partitionné en deux. La partie statique, que l'on nomme *architecture du métaniveau*, décrit les éléments du système (ses structures et ses procédures) et comment ils sont assemblés. La partie dynamique, qui est décrite en terme de *protocoles*, explicite la manipulation des éléments du système qui est la sémantique de l'exécution du langage. On fait en CLOS la distinction entre le langage lui-même et le métaniveau.

Le MOP fournit au métaprogrammeur (qui dans notre cas correspond au fournisseur de services) un mécanisme de contrôle pour étendre et particulariser le comportement par défaut de CLOS (cf fig 4.2). Ces extensions se font en créant des sous-classes des classes du MOP.

Le métaprogrammeur fournit à l'utilisateur de services des métaclasses qui spécialisent les classes du MOP. L'utilisateur de services pour intégrer un service dans son code instancie ces classes. Comme dans les intergiciels et à la différence des plateformes à composants, le code de l'application est lié (par le fait d'être instancié par une métaclasse) au code technique. L'aspect déclaratif du paramétrage de l'intégration de services n'est plus possible.

L'intégration d'un service de persistance issu de la littérature [91, 38, 90] s'écrit comme suit. Pour intégrer un service de persistance il faut sous-classer les métaclasses standard qui ont trait à l'accès aux variables d'instances (notamment `slot-value`) et surcharger leurs méthodes. De plus les optimisations pour les accès aux variables d'instances doivent être désactivées par la métaclasse de persistance, car les variables d'instances peuvent être temporaires ou persistantes et aucune décision ne peut être prise à la compilation. Il faut modifier le comportement de la métaclasse qui gère la définition des variables d'instances pour ajouter l'option de variables d'instances `temporaire`. Pour cela il faut masquer les méthodes impliquées pendant l'exécution de la forme `defclass` pour valider cette option de variables d'instances.

La complexité d'intégration pour le fournisseur de services est forte. Il faut comprendre et maîtriser l'impact du sous-classage des métaclasses standard dans le sys-

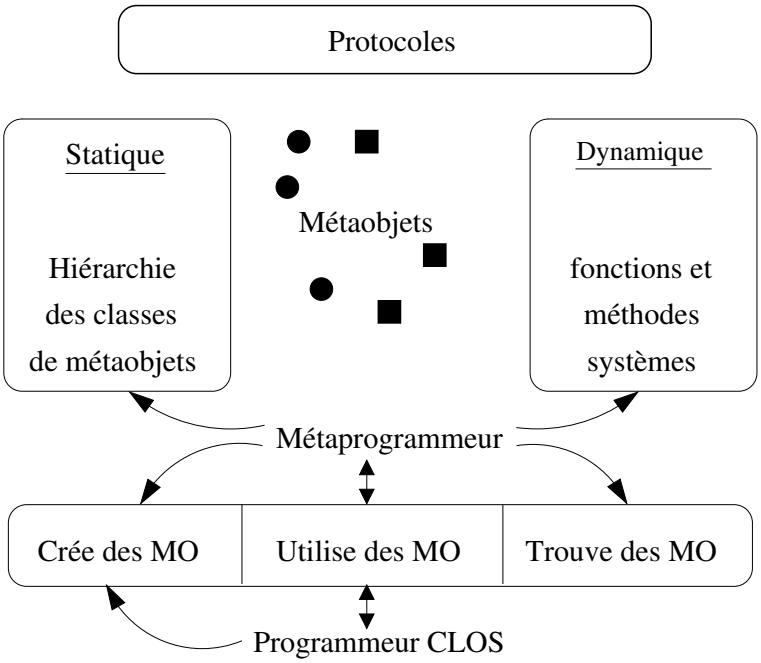

FIG. 4.2 – CLOS : Séparation programmeur / métaprogrammeur

tème. La complexité augmente quand il faut composer plusieurs services, il faut vérifier leur compatibilité quand l'utilisateur de services va faire hériter ses classes de différentes métaclasses [13]. L'utilisateur du service ne peut pas facilement adapter le service sans modifier le métaprogramme. Les métaclasses sont très dépendantes du système dans lequel elles sont implémentées et ne sont pas portables.

4.1.2 Description de la communication entre objets

Nous avons vu que les protocoles à métaobjets ouvrent des points d'entrées dans la structure et l'exécution du système. D'autres approches à base de métaobjets telles que CODA [2] et Actalk [14] décrivent et permettent de contrôler les communications entre objets. Ces approches simplifient et offrent une meilleure abstraction de la métaprogrammation. Ces approches réifient par des métaobjets le comportement d'un objet comme l'exécution, l'envoi de message, la liaison d'un message avec une méthode, et le changement d'état. La modification du comportement est effectuée comme dans les protocoles à métaobjets en redéfinissant les comportements (cf. figure 4.3).

CODA, à la différence des approches comme celle de CLOS, n'expose pas des points de construction du langage mais des points plus abstraits qui ont trait à la sémantique d'exécution. CODA est plus indépendant du langage que le MOP de CLOS mais moins précis. De plus la définition d'un comportement est localisée à l'intérieur d'un métaobjet et non plus éparpillée dans différents protocoles d'un MOP.

L'intégration d'un service à l'aide de CODA est similaire à l'intégration d'un service avec CLOS, sauf que les structures manipulées sont beaucoup plus simples et beaucoup plus abstraites. L'idée d'abstraction du modèle d'exécution réduit la complexité de mise en œuvre des services. CODA implémente ce métaniveau (constitué des métaobjets) entre les objets, ce qui est un frein à la portabilité et entraîne beau-

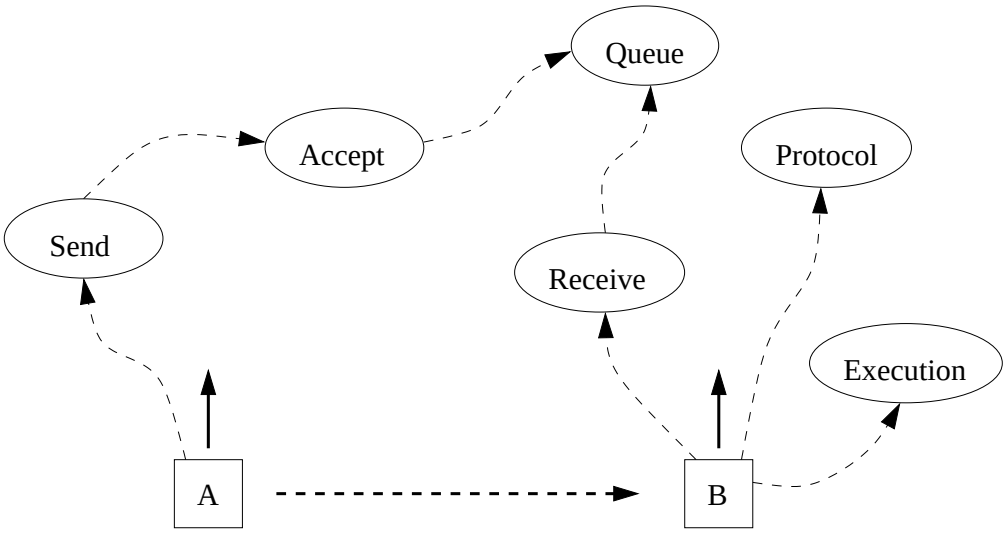

FIG. 4.3 – Les métaobjets de CODA

coup d'indirections. D'autre part la composition de plusieurs services ainsi que la parramétrisation des services restent difficiles.

4.1.3 Récapitulatif de l'intégration de services par la métaprogrammation

La métaprogrammation permet de spécialiser des points d'entrées prédéfinis pour intégrer des services, mais l'écriture et la mise en œuvre sont complexes pour le fournisseur de services. La composition doit être gérée manuellement et il est difficile d'adapter l'intégration d'un service pour l'utilisateur de services.

Les points forts de ces approches sont la notion de points d'entrées qui est reprise dans la programmation par aspects (cf. section 4.2 et la notion d'abstraction de la sémantique d'exécution dans CODA). La difficulté majeure reste la complexité de mise en œuvre.

La table 4.1 résume les propriétés d'intégration de services grâce à la métaprogrammation.

4.2 Les approches par tissage

Nous regroupons sous le terme d'approche par tissage, la programmation par aspects et les langages dédiés qui sont des sous-domaines de la programmation générative [28]. Toutes ces approches ont en commun la séparation du code de l'application et du code technique qui est écrit dans un langage spécifique. L'application finale est générée à l'aide d'un outil nommé *tisseur* (cf. figure 4.4).

Sur les bases de la métaprogrammation, ces approches utilisent le concept des points d'entrées, nommé *points de jointures*, pour modifier le comportement de l'application. La différence fondamentale avec la métaprogrammation est l'utilisation de langages spécifiques souvent déclaratifs qui facilitent l'écriture du code technique. Nous passons en revue plusieurs approches qui se différencient par les points de jointures qu'elles exposent, par l'expression et la facilité de manipulation de ces points de

Critère	Évaluation
Gestion de l'hétérogénéité	**mauvaise** pour les MOP et **moyenne** pour CODA à cause de l'implantation physique du métaniveau
Séparation des services	**bonne** dans la définition mais **difficile** car la composition doit être faite à la main
Adaptabilité de l'intégration	**mauvaise** car tout est câblé dans le métaprogramme
Découplage fournisseur / utilisateur	**bon** car l'utilisateur n'a qu'à instancier ses classes à partir d'une métaclasse
Complexité de mise en œuvre	**forte** pour le fournisseur et **faible** pour l'utilisateur

TAB. 4.1 – Résumé des propriétés d'intégration de services à l'aide de la métaprogrammation

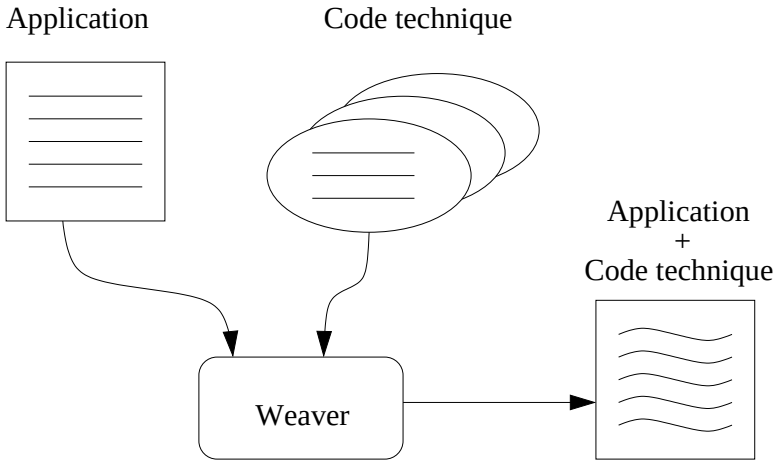

FIG. 4.4 – Mécanisme général des approches par tissage

jointures et par la composition des comportements définis sur des points de jointures identiques.

4.2.1 La programmation par aspects

La programmation par aspects [49], dont AspectJ [48] est un des représentants, est issue des travaux de Gregor Kiczales autour de la séparation des préoccupations (on trouve des prémisses de ces approches dans les langages de modèles de connaissances [60]). Les éléments clés de la programmation par aspects sont le modèle de point de jointure, la définition des comportements et la modification des structures. Les points de jointures sont des points d'exécution sur lesquels du comportement est ajouté. La modification des structures permet de modifier les définitions des classes.

AspectJ offre un certain nombre de points de jointure tels que l'envoi d'un message, l'exécution d'une méthode, l'envoi d'une exception. Pour décrire les possibilités du langage AspectJ pour l'intégration de services nous prenons l'exemple du patron de conception observateur/observable issu de [39] qui est équivalent à un service de notification.

Dans cet exemple, l'aspect `SubjectObserverProtocol` déclare un point de jointure nommé `stateChanges` qui capture l'exécution de la méthode `click` des objets `Button`. La condition d'exécution décrit qu'après le passage dans le point de jointure `stateChanges` la méthode `update` est appelée sur l'ensemble des observateurs de l'observable. La classe de l'observable est modifiée pour que les instances disposent d'une liste de ses observateurs et des méthodes d'ajout et de retrait d'observateurs. Et la classe des observateurs est modifiée pour que les instances aient un champ qui pointe sur l'observable et des méthodes d'accès et de modification de ce champ.

```
1  aspect SubjectObserverProtocol {
2      pointcut stateChanges(Subject s):
3          target(s) &&
4          call(void Button.click());
7      after(Subject s): stateChanges(s) {
8          for (int i = 0; i < s.getObservers().size(); i++) {
9              ((Observer)s.getObservers().elementAt(i)).update();
10         }
11     }
13     private Vector Subject.observers = new Vector();
14     public void   Subject.addObserver(Observer obs) {
15         observers.addElement(obs);
16         obs.setSubject(this);
17     }
18     public void   Subject.removeObserver(Observer obs) {
19         observers.removeElement(obs);
20         obs.setSubject(null);
21     }
22     public Vector Subject.getObservers() { return observers; }
```

```
24      private Subject Observer.subject = null;
25      public void     Observer.setSubject(Subject s) { subject = s; }
26      public Subject  Observer.getSubject() { return subject; }
27  }
```

Les aspects écrits en AspectJ sont très liés aux classes de l'application, même avec la notion d'aspect abstrait. Il n'y pas de découplage fort entre le fournisseur de services et l'utilisateur de services car la définition des points de jointures est liée aux classes de l'application. Leurs rôles se confondent en AspectJ.

Nous avons vu qu'un composant est en réalité une composition de plusieurs objets et que le code technique est distribué à l'intérieur de ces objets. Du fait qu'AspectJ soit très proche du langage cible et que le grain de définition soit la classe, il est difficile de garder la granularité des composants dans la définition des aspects. Pour poser un aspect sur un composant, il faut poser des aspects sur les différentes classes des objets qui forment le composant. Le résultat immédiat est qu'il est impossible d'utiliser ce même aspect sur un composant dans une plateforme à composant différente. En effet nous avons vu que les différentes implantations des plateformes à composants utilisent des configurations d'objets différentes pour faire exécuter un composant.

Quand deux aspects ajoutent du comportement sur le même point de jointure, AspectJ séquentialise les deux comportements suivant un ordre dans lequel intervient l'ordre des déclarations. L'utilisateur de services doit donc prendre garde quand il utilise plusieurs aspects que le comportement résultant de la composition reste cohérent.

Dans les paragraphes suivants nous décrivons d'autres représentants de la programmation par aspects, qui offrent des points de variations par rapport à AspectJ.

L'approche *EAOP* [34] propose d'utiliser des aspects qui sont déclenchés par des successions d'événements émis pendant l'exécution d'une application. Cette approche permet de modéliser des points de jointures plus fins qu'AspectJ avec un meilleur pouvoir expressif et expose la composition à l'utilisateur de services.

Les approches *Aspectual Collaborations* [53] et *Caesar* [59] sont aussi des approches de la programmation par aspects qui offrent de meilleures capacités de modularisation et de réutilisation.

JAC [95, 94], qui est à la fois un framework d'aspects et un serveur d'applications, propose d'écrire les aspects dans le même langage que le langage de l'application. Il offre la possibilité de poser les aspects sur l'application dynamiquement et permet le développement de la configuration des aspects dans son serveur d'application.

JBossAOP [16] et *AspectJ2EE* [23] sont des langages d'aspects qui s'intéressent respectivement aux plateformes à composants JBoss et WebSphere qui sont des implantations de la spécification EJB. Ces approches sont spécifiques à une implantation précise d'une plateforme à composants et offrent des constructions similaires à celles des approches présentées précédemment.

Les *mixins* [37] ainsi que les traits [99] tout en étant dans les mêmes perspectives que la programmation par aspects sont des extensions des modèles à objets. La définition d'une mixin (ou d'un trait) se fait dans le langage de l'application. Une mixin définit du comportement qui est `mixé` dans les classes à modifier. Les traits apportent en plus la notion de propriétés fournies et requises.

La *programmation orientée sujet* [89] ouvre la composition à l'utilisateur de services en le laissant écrire et choisir les règles de composition qui s'appliquent entre les différents sujets.

ISL [7] offre un modèle de composition automatique des aspects écrit dans le langage ISL. Cela permet de définir des aspects totalement indépendants les uns des autres tout en ayant un résultat de la composition des aspects qui soit déterministe et la détection d'erreurs.

À un niveau plus formel on trouve la *superposition* [54] qui propose une solution au problème de la composition des comportements. La superposition est basée sur la catégorisation de Goguen dans la théorie des systèmes généraux. La superposition permet de composer et de raffiner des comportements à l'aide d'un langage mathématique.

4.2.2 Les langages dédiés

Les langages dédiés [108] sont des langages de programmation dédiés à la résolution d'un problème particulier. Ce sont de petits langages,souvent impératifs, moins expressifs que les langages généraux mais plus concis et plus lisible qui fournissent des abstractions et des notations appropriées au domaine d'application. Ces langages sont parfois considérés comme des langages de spécification car ils sont déclaratifs et cachent les détails d'implantation. Les langages dédiés capturent l'expertise soit implicitement en cachant les patterns de programmation communs dans l'implantation soit explicitement en exposant des points de paramétrage.

Du point de vue des architectures logicielles les langages dédiés peuvent être vus comme un mécanisme de paramétrage et comme un modèle d'interface. Il est possible de considérer un programme d'un langage dédié comme un argument complexe d'un programme hautement paramétrable. Un langage dédié peut aussi offrir une interface dédiée à une librairie générique en utilisant des patterns d'appels communs.

Un autre avantage des langages dédiés est qu'ils permettent de faire de la vérification car leur sémantique est plus restreinte que celle des langages généraux et permettent la décidabilité de certaines propriétés critiques d'un domaine.

Dans le cadre de l'intégration de services un langage dédié défini un langage d'intégration d'un service (comme par exemple le langage `D` pour la distribution [55], ou `Teapot` pour la cohérence de la réplication de données [19]). L'avantage d'une telle approche est que le langage est parfaitement ciblé pour la description de l'intégration d'un service. L'utilisateur du service écrit un programme dans le langage dédié, qui correspond aux paramètres de l'intégration du service dans son application. Le degré de paramétrage est fort et le code produit est souvent optimum. Le fournisseur de services doit construire un parser pour lire les programmes du langage dédié et un générateur de code ce qui est assez complexe. De plus les problèmes de composition de l'application de plusieurs langages dédiés ne sont absolument pas traités par cette approche.

On notera aussi l'existence de XAspects [100] qui combine l'approche de la programmation par aspects et des langages dédiés, en permettant de créer des langages dédiés au dessus de AspectJ.

4.2.3 Récapitulatif de l'intégration de services par les approches par tissage

Les approches par tissage reprennent la notion de points d'entrée de la métaprogrammation tout en facilitant la déclaration et l'ajout de nouveaux comportements. La difficulté majeure pour l'intégration de services dans les plateformes à composants avec ces approches est qu'elles ne prennent pas en compte qu'un composant est une composition d'objets et que ces compositions d'objets sont hétérogènes dans les différentes implantations de plateformes. Le pouvoir expressif de ces approches se concentre sur la notion d'objet ou de classe. La séparation des services est bonne parce qu'ils sont définis de manière indépendante les uns des autres que ce soit dans des aspects ou dans des langages dédiés différents. Mais la composition est souvent difficile car laissée à la charge de l'utilisateur de services sauf dans l'approche ISL qui automatise la composition. L'adaptabilité de l'intégration est variable suivant le degré de paramétrage offert par l'approche. L'ensemble des propriétés d'intégration de services des approches par tissage est résumé dans le tableau 4.2.

Critère	Évaluation
Gestion de l'hétérogénéité	**mauvaise** car ces approches se basent sur la notion de classe dans un langage donné et non de composant
Séparation des services	**bonne** dans la définition mais **difficile** car la composition doit être faite à la main (sauf pour ISL)
Adaptabilité de l'intégration	**variable** car les approches offrent différentes possibilités de paramétrage
Découplage fournisseur / utilisateur	**bon** car l'utilisateur n'a qu'à instancier les aspects sur ses classes
Complexité de mise en œuvre	**moyenne** pour le fournisseur et **faible** pour l'utilisateur

TAB. 4.2 – Résumé des propriétés d'intégration de services à l'aide des approches par tissage

4.3 Les approches par modélisation

Les approches par modélisation tentent de répondre au problème de la décomposition du code des applications et du code technique au niveau du modèle pour s'abstraire des problèmes que nous venons de voir.

La modélisation orientée sujet [20, 22] part de la constatation que dès la modélisation des applications il y a une différence de structure entre la description des propriétés d'une application et sa représentation dans un langage de modélisation. Cette différence de structure fait que certaines propriétés sont divisées et réparties dans différents éléments du modèle. Ces propriétés sont dans notre cas les appels vers les services.

Pour résoudre ce problème la modélisation orientée sujet étend la modélisation orientée objet en lui ajoutant des capacités de décomposition additionnelles. Cette

approche supporte la modélisation des différentes propriétés dans différents modèles qui peuvent se surcharger. La composition de ces différents modèles est spécifiée par des relations de compositions. Une relation de composition identifie des éléments (nommé des éléments correspondants) se surchargeant dans différents modèles, et spécifie comment les éléments correspondants doivent être réconciliés. Plusieurs types de réconciliation sont possible comme par exemple une réconciliation par écrasement (une propriété est prépondérante), ou une réconciliation par fusion (il faut alors fondre deux propriétés en une seule).

Cette approche permet une spécification de l'intégration de services au niveau du modèle en permettant à l'utilisateur de services de choisir la composition des éléments se surchargeant. La composition des services se fait donc manuellement. De plus cette approche reste au niveau modèle et n'incorpore pas la notion de composant. L'adaptation de l'intégration d'un service se fait en modifiant le modèle qui le définit, ce qui peut entraîner de la complexité pour l'utilisateur de services.

Une autre approche de l'intégration de services par la modélisation est celle proposée par l'OMG au travers de UML 2 et des *profils* UML [79, 72]. Cette évolution d'UML et du MOF se place dans le cadre de la démarche de l'architecture dirigée par les modèles MDA (Model Driven Architecture) [17, 87]. Le processus MDA permet de séparer les spécifications fonctionnelles d'un système de spécifications de son implantation sur une plateforme donnée. La MDA définit plusieurs niveaux d'architectures. La version initiale de référence (cf. figure 4.5) expose un modèle indépendant des plateformes qui après raffinement se projette dans un modèle spécifique à une plateforme.

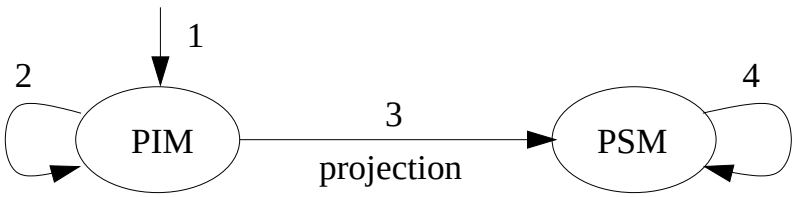

FIG. 4.5 – Processus MDA

Un des avantages de cette approche est de définir une architecture indépendante pour pouvoir ensuite la projeter vers différents modèles de plateformes. La prise en compte de ces différents modèles de plateformes au sein d'UML se fait grâce aux *profils*. Le package des *profils* contient des mécanismes qui permettent aux métaclasses des métamodèles existants d'être étendus en les annotant. Cela permet de modifier les différents métamodèles d'UML pour les différentes plateformes telles que EJB ou CCM. Le mécanisme des profils est compatible avec le MOF.

La mise en œuvre de la MDA repose sur les systèmes de transformation entre modèles. Les solutions jusqu'à présent sont des transformations ad hoc, en fonction des besoins. Pour pallier cela l'OMG travaille sur la spécification d'un modèle de transformations nommé QVT (Queries Views Transformations) [73]. Les *Queries* prennent en entrée un modèle et sélectionne des éléments spécifiques de ce modèle. Les *Views* sont des modèles qui sont dérivés d'autres modèles, et les *Transformations* prennent en entrée un modèle et le mettent à jour ou créent un nouveau modèle. Deux types de transformations sont définis par QVT, les relations et la projection. La première

permet de vérifier la cohérence entre deux modèles, tandis que le mapping définit des liaisons unidirectionnelles qui génèrent en retour une valeur (un modèle).

L'avantage principal de cette approche est la prise en compte des différents modèles de composants. La modélisation est indépendante et donc réutilisable. Et grâce à un système de projections, les différentes plateformes à composants sont atteintes. Mais pour le moment l'approche MDA reste floue, la spécification des transformations n'est pas terminée. A l'heure actuelle les transformations sont réalisées de manière ad hoc comme par exemple [6, 97]. De plus les définitions des modèles de plateformes à composants sont basées sur les spécifications de ces modèles et non sur les implantations ce qui ne convient pas à l'intégration de services comme nous l'avons vu dans le chapitre 3.

4.3.1 Récapitulatif de l'intégration de services par les approches par modélisation

Les approches par modélisation permettent de définir l'intégration de services à un niveau indépendant des plateformes à composants qui permet de porter les définitions d'intégration de services entre les différentes implantations de plateformes à composants. La difficulté majeure pour le moment reste les systèmes de transformations qui sont peu développés et souvent de manière ad hoc. La modélisation des implantations de plateformes à composants n'est pas complètement prise en compte pour le moment dans ces approches. La définition d'un métamodèle global de plateforme reste un verrou technologique. La composition des services est laissée à la charge de l'utilisateur de services. Elle est facilitée dans l'approche de la modélisation orientée sujet par la spécification de relations de composition. L'adaptabilité des services se fait par manipulation du modèle du service, ce qui induit de la complexité pour l'utilisateur de services. Le découplage entre le fournisseur et l'utilisateur n'est pas toujours clair dans ces approches, souvent le fournisseur joue aussi le rôle de l'utilisateur quand il faut intégrer plusieurs modèles. L'ensemble des propriétés d'intégration de services des approches par modélisation est résumé dans le tableau 4.3.

Critère	Évaluation
Gestion de l'hétérogénéité	**bonne** car la définition des intégrations de services se fait indépendamment des plateformes à composants. Mais les transformations restent un problème
Séparation des services	**bonne** dans la définition mais **difficile** car la composition doit être faite à la main
Adaptabilité de l'intégration	**moyenne** car elle faite au niveau de la modélisation mais pas de paramétrage
Découplage fournisseur / utilisateur	**faible** car c'est souvent le fournisseur qui a la connaissance pour l'intégration des modèles
Complexité de mise en œuvre	**faible** pour le fournisseur et **faible** pour l'utilisateur

TAB. 4.3 – Résumé des propriétés d'intégration de services à l'aide des approches par modélisation

Conclusion

Nous avons vu dans ce chapitre, différentes techniques qui permettent de réaliser l'intégration de services. Les deux premières approches (métaprogrammation et tissage) ont montré l'intérêt de réifier des points d'exécution du programme pour la description de l'intégration de services. Les approches par tissage facilitent cette description en proposant un support plus explicite de la définition des points de jointure et de la mise en rapport de ces points de jointures avec les services qui y sont attachés. Le problème majeur de ces approches est la composition qui est peu expressive. La seule approche qui réponde à ce besoin est ISL. L'autre problème de ces approches est qu'elles ne prennent pas en compte la structure des composants et sont trop proches des modèles à objets. Les approches par modélisation permettent une meilleure abstraction de la description de l'intégration de services en utilisant des modèles indépendants des plateformes à composants. Mais pour l'instant, les systèmes de transformation souvent ad hoc sont difficiles à gérer.

Récapitulatif

Dans cette partie nous avons étudié l'intégration de services sous différentes formes. Dans un premier temps (cf. chapitre 2) nous avons étudié des services normalisés par l'OMG et de leur intégration dans l'intergiciel de communication CORBA. Nous avons constaté que l'intégration de services de manière manuelle est complexe et répétitive pour l'utilisateur de services ainsi que pour le fournisseur de services.

Dans un deuxième temps (cf. chapitre 3) nous avons étudié les plateformes à composants qui offrent l'intégration automatique des services. L'utilisateur ne fournit qu'un descripteur de déploiement dans lequel il donne les valeurs des paramètres des services. Mais les plateformes à composants que nous avons étudiées ne s'intéressent pas au fournisseur de services qui veut rajouter de nouveaux services. En effet les plateformes se basent sur des générateurs pour produire le code des conteneurs qui vont contrôler l'exécution des composants. De plus l'hétérogénéité des plateformes à composants ne permet pas de porter l'ajout d'un nouveau service d'une plateforme à l'autre.

Dans un troisième temps (cf. chapitre 4) nous avons étudié différentes techniques de séparation des préoccupations pouvant aider le fournisseur de services à réaliser l'intégration de services. Nous avons vu que la métaprogrammation permet de redéfinir le comportement de certains points d'entrées du langage. Mais cette redéfinition est complexe car il faut comprendre et maîtriser les points sémantiques ouverts par la métaprogrammation. L'approche CODA utilise des points d'entrées conceptuels à la place des points sémantiques. Cela facilite largement la définition des intégrations de services. Nous avons étudié les approches par tissage qui offre des facilités dans la description des intégrations de services avec leurs langages déclaratifs, mais restent trop proches des langages et ne permettent pas de résoudre le problème de l'hétérogénéité des plateformes. Nous avons aussi étudié les approches par modélisation qui permettent de s'abstraire des plateformes à composants. Elles présentent des qualités d'indépendance et de réutilisation mais sont difficiles à mettre en œuvre à cause du manque de support pour les systèmes de transformations.

Dans la partie suivante nous présentons notre modèle pour l'intégration de services dans les plateformes à composants. En nous servant de l'étude précédente, nous proposons une approche basée sur un modèle indépendant des plateformes qui offre

des points de jointure décrivant de manière conceptuelle l'exécution d'un composant. Ce modèle offre des propriétés de composition et de détection d'erreurs et permet le paramétrage de l'intégration de services.

Deuxième partie

UNE APPROCHE MDA DE L'INTÉGRATION DE SERVICES

Dans cette partie...

En nous basant sur l'étude bibliographique (cf. partie I), nous définissons un modèle d'intégration de services qui répond aux problèmes de l'hétérogénéité des plateformes à composants et de la composition des services.

Nous commençons par développer un scénario d'intégration de services qui présente notre approche dans sa globalité (cf. chapitre 5). Nous définissons deux exemples simples qui nous permettent d'argumenter les différents choix de notre approche.

Ensuite nous présentons le modèle des descriptions abstraites de composants qui représentent les composants indépendamment des plateformes à composants (cf. chapitre 6). Nous décrivons les opérations d'évolution structurelle et comportementale ainsi que le modèle des intégrateurs qui permettent de paramétriser les descriptions d'intégration de services. Le modèle des descriptions abstraites de composants ainsi que celui des intégrateurs reposent sur des systèmes externes sur lesquels nous imposons des contraintes.

Puis nous formalisons l'étape de composition des évolutions structurelles et comportementales (cf. chapitre 7). Nous prouvons que le système de composition fournit un résultat unique et déterministe quelque soit l'ordre d'application des évolutions. Cette phase de composition permet de détecter des conflits entre les évolutions.

Enfin nous présentons le modèle des composants concrets qui permet de combiner les évolutions structurelles et comportementales aux informations issues de la plateforme à composants cible (cf. chapitre 8). Nous détaillons le processus de raffinement qui mène jusqu'à la projection du code dans la plateforme à composants cible. Cette phase permet de détecter des conflits entre les évolutions et la plateforme à composants cible.

CHAPITRE 5

Notre scénario de l'intégration de services

Dans ce chapitre nous détaillons notre approche de l'intégration de services à l'aide d'un scénario. Ce chapitre expose, à l'aide d'exemples simples, l'ensemble des éléments et les différentes phases de notre approche de l'intégration de services qui sont détaillés dans les chapitres suivants.

Comme nous l'avons fait dans la bibliographie nous séparons la vision du fournisseur de services de celle de l'intégrateur de services. Dans un premier temps nous nous intéressons au fournisseur de services qui doit décrire comment intégrer son service dans des composants métier et fournir les composants de services. Puis nous passons au configurateur de l'application qui doit paramétrer les descriptions d'intégration de services, et donner la configuration générale de l'application. Enfin nous étudions les différentes étapes du processus d'intégration de services que sont la décoration, la composition, la correspondance et la génération de code. Nous développons deux exemples volontairement simples pour ne pas surcharger le discours. Des exemples plus complexes sont développés dans la partie III.

5.1 Le fournisseur de services

Dans notre approche le fournisseur de services doit fournir pour le service qu'il veut intégrer d'une part les composants de services et d'autre part une description, que l'on nomme *intégrateur*, des évolutions à apporter aux composants métier pour qu'ils puissent interagir avec les composants de services (cf. figure 5.1).

5.1.1 Composants de services

Le travail du fournisseur de services est tout d'abord de construire des composants qui vont réaliser la logique métier du service. Ce sont des composants applicatifs comme les autres.

Exemple :
Dans un service de notification, le fournisseur de services doit fournir le canal dans lequel on va poster les notifications d'arrivée de messages. Dans un autre cas comme par exemple un service de statistiques de comptage d'appels, il peut ne pas y avoir de composant de service. Tout se fait par des transformations du composant métier.

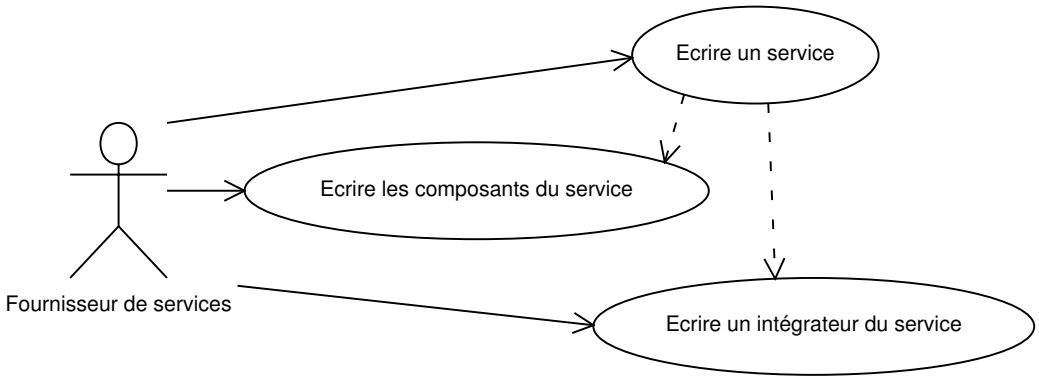

FIG. 5.1 – Ecriture d'un service et de son intégrateur

Plusieurs politiques sont possibles pour ces composants de services à l'intérieur d'une plateforme à composants. Par exemple on peut avoir une seule instance d'un composant de service pour la plateforme ou avoir un composant de service pour chaque composant métier qui utilise ce service. Il faut donc pouvoir ajuster ces politiques en influant sur l'infrastructure de déploiement et le cycle de vie des composants eux-mêmes.

5.1.2 Intégrateurs

Le fournisseur de services doit ensuite décrire les évolutions à apporter aux composants qui veulent utiliser les composants du services. Les descriptions des évolutions s'appuient sur un modèle qui abstrait la notion de composant. Un composant ainsi que l'ensemble des objets de gestion qui l'entourent est représenté dans notre modèle par une *description abstraite de composant* (DAC). Nous regroupons les descriptions des évolutions pour l'intégration d'un service dans ce que nous appelons un *intégrateur* dans notre modèle.

Comme nous l'avons vu précédemment un des problèmes de l'intégration de services est l'hétérogénéité des plateformes à composants. Dans notre approche, à l'aide du modèle de descriptions abstraites de composants et des intégrateurs qui se basent sur ces descriptions abstraites de composants, nous définissons l'intégration de services de manière indépendante des plateformes à composants.

L'étude bibliographique a mis en évidence que l'intégration de services ne consiste pas uniquement à décrire des évolutions structurelles du composant. Il faut aussi pouvoir décrire des évolutions comportementales. Notre modèle de descriptions abstraites de composants permet de modéliser des évolutions structurelles et comportementales. Les évolutions structurelles permettent d'ajouter des propriétés (attributs et méthodes). Tandis que les évolutions comportementales permettent de modifier les émissions et réceptions de requêtes des composants.

Le modèle des intégrateurs (cf. section 6.5) est un modèle qui se base sur le modèle des descriptions abstraites de composants (cf. section 6.1). Le modèle des intégrateurs permet la manipulation et le paramétrage des descriptions abstraites de composants.

Les modèles de descriptions abstraites de composants et le modèle des intégrateurs tels que définis dans le chapitre 6 sont indépendants des systèmes de typage,

visibilités, propriétés et règles. Pour pouvoir être utilisés par le modèle de descriptions abstraites de composants et le modèle des intégrateurs, ces différents systèmes doivent répondre à certaines contraintes (comme par exemple renvoyer un type commun entre deux types pour le système de typage s'il existe ou bien le type nul) (cf. chapitre 6).

Dans ce chapitre nous utilisons les mêmes systèmes de typage, visibilités, propriétés et règles que ceux définis et utilisés dans la partie application (cf. partie III). Nous ne les détaillons pas ici mais nous donnons les informations nécessaires pour comprendre les exemples au fur et à mesure.

Le modèle de descriptions abstraites de composants permet, pour les évolutions comportementales, de modéliser différents types de flots de requêtes entre les composants. Dans les exemples qui suivent ainsi qu'en partie III nous utilisons un modèle qui représente un flot RPC (Remote Procedure Call). Cette modélisation est faite par un ensemble de métaobjets qui représentent des étapes dans le flot d'une requête entre deux composants (cf. figure 5.2) et est issue des travaux autour de CODA [2].

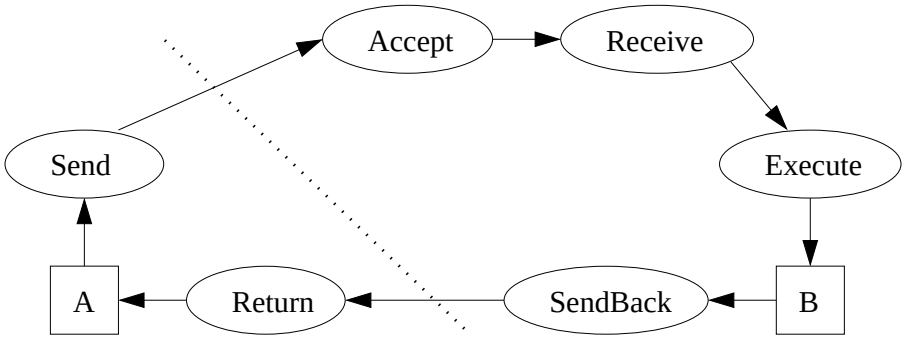

FIG. 5.2 – Configuration de métaobjets qui représentent un flot RPC entre deux composants A et B

Le flot de requêtes est décrit dans un intégrateur que l'on nomme habituellement : *plan de base*. Ce flot de requêtes est décrit sous la forme d'évolutions comportementales. L'intégrateur du plan de base prend en paramètre la description abstraite de composant. Dans la figure 5.3 nous montrons un intégrateur qui représente le plan de base du flot RPC. Ce plan de base est constitué de six métaobjets qui pour toutes les méthodes (désignées ici par une étoile '*'), quelque soit leur visibilité (désignée ici par deux étoiles '* *'), font transiter le message. Nous utilisons ici le système de règles de réécriture ISL [4] pour déclarer les évolutions comportementales. Dans cet intégrateur chaque règle déclare son comportement standard, c'est le `_call`, puis fait suivre le message au métaobjet suivant.

Exemple :

La figure 5.4 montre l'intégrateur du service de notification. Cet intégrateur s'applique sur une description abstraite de composant, et prend en paramètre un composant qui implémente l'interface `Channel`. Il ajoute un attribut de service pour stocker le canal et une méthode de service pour changer la valeur de cet attribut. Il possède une évolution comportementale qui signifie qu'après chaque appel de méthode reçue (le métacaractère `'*'` désigne l'ensemble des signatures de méthodes), la signature de la méthode appelée est ajoutée dans le canal. Cette transformation ne s'applique que sur les signatures de méthodes *publiques* associées à la portée *métier* du composant.

```
Integrator PlanDeBaseRPC (DAC d){
  [d, * *, *, Send(Message m)] :-
      NetworkRequest n := _call(m) ; Accept(n)
  [d, * *, *, Accept(NetworkRequest n)] :-
      Message m := _call(n) ; Receive(m)
  [d, * *, *, Receive(Message m)] :-
      Message m1 := _call(m) ; Execute(m1)
  [d, * *, *, Execute(Message m)] :-
      Message m1 := _call(m) ; SendBack(m1)
  [d, * *, *, SendBack(Message m)] :-
      NetworkRequest n := _call(m) ; Return(n)
  [d, * *, *, Return(NetworkRequest n)] :- _call(n)
}
```

FIG. 5.3 – Intégrateur du plan de base RPC

```
Integrator Notification (DAC d, chan implements Channel){
  [d, service metier] Channel c = new chan();
  [d, service metier] void setChannel(Channel c1) {c = c1}
  [d, public metier, *, Receive(Message m)] :- _call(m) ; c.push(m)
}
```

FIG. 5.4 – Intégrateur du service de notification

Exemple :

L'intégrateur figure 5.5 est l'intégrateur d'un service de comptage d'appels reçus. Dans cet exemple on ajoute à une description abstraite de composant une variable de service qui va permettre de stocker le nombre de méthodes exécutées. On ajoute aussi un accesseur pour cette variable. Enfin on modifie le comportement des méthodes de la description abstraite de composant pour qu'après chaque appel de méthode reçue, le compteur soit incrémenté.

```
Integrator StatAppelsRecus (DAC d){
  [d, service metier] long nbCall = 0;
  [d, public metier] long nbReceivedCall() {return nbCall}
  [d, public metier, *, Receive(Message m)] :- _call(m) ; nbCall++
}
```

FIG. 5.5 – Intégrateur du service de statistiques

Ces différents modèles forment un modèle général de contrôle de l'intégration de services (cf. figure 5.6). Ce modèle de contrôle d'intégration de services se place dans la lignée des travaux sur l'ingénierie de modèles dans la mesure ou il s'abstrait des modèles de types de plateformes, des modèles de règles de spécifications de comportements, et des modèles de systèmes de types. Cela permet de définir différents modèles d'intégrateurs.

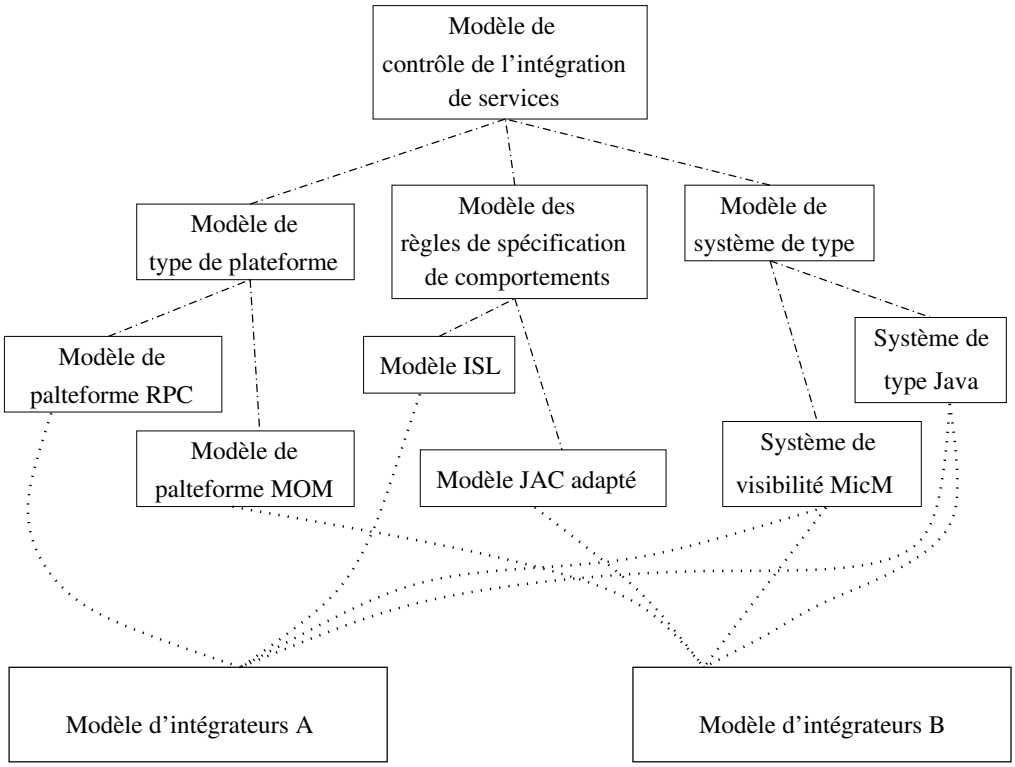

FIG. 5.6 – Du modèle de contrôle de l'intégrations de services aux modèles d'intégrateurs

5.2 Le configurateur d'applications

Le rôle du configurateur d'applications est d'ajouter les services à l'application (c'est l'utilisateur des services). Dans notre approche le configurateur d'applications va utiliser les intégrateurs des services qui correspondent aux services qu'il veut intégrer dans l'application et à l'aide d'un fichier de configuration va instancier et paramétrer ces intégrateurs (cf. figure 5.7). Le configurateur d'application va lancer le processus d'intégration et le guider à travers les différentes étapes. Il va aussi résoudre les conflits qui peuvent surgir lors des différentes phases de ce processus d'intégration.

Le configurateur écrit dans le fichier de configuration l'association entre les intégrateurs et les paramètres que sont les composants de son application. La vérification du typage sur les paramètres est faite dans la phase de correspondance (cf. chapitre 8). Dans ce fichier le configurateur d'applications précise quel est le type de flot de requêtes utilisé. Le configurateur démarre ensuite le processus d'intégration de services.

`Exemple :`

Pour les services de notification et de statistique, nous présentons un fichier de configuration (cf. figure 5.8) pour intégrer ces deux services dans un composant de nom `MonComposantTest`. Dans ce fichier de configuration nous utilisons les deux intégrateurs `Notification` et `StatAppelsRecus` et l'intégrateur pour le plan de base RPC. Ces deux intégrateurs sont appliqués au composant de nom `MonComposantTest`. Pour l'intégrateur

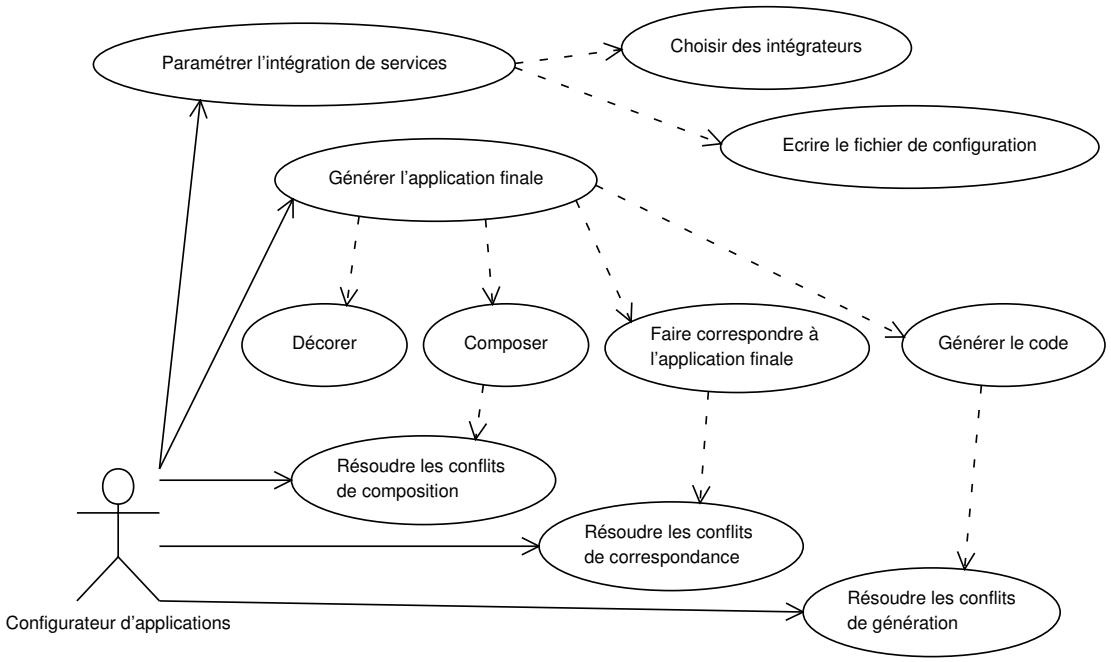

FIG. 5.7 – Configuration des services et génération de l'application

`Notification` nous passons en plus une classe implémentant l'interface `Channel`.

```
<Config>
   <PlanDeBaseRPC>
      <d> MonComposantTest </d>
   </PlanDeBaseRPC>
   <Notification>
      <d> MonComposantTest </d>
      <chan> fr.essi.rainbow.ChannelImpl </chan>
   </Notification>
   <StatAppelsRecus>
      <d> MonComposantTest </d>
  </StatAppelsRecus>
</Config>
```

FIG. 5.8 – Fichier de configuration pour la notification et les statistiques

5.3 Étapes du processus d'intégration de services

Le processus d'intégration se décompose en quatre grandes étapes que sont la décoration, la composition, la correspondance et la génération. Les deux premières étapes sont indépendantes des composants de l'application et utilisent le modèle des descriptions abstraites de composants qui abstrait la notion de composant par rapport à une plateforme cible. À partir de l'étape de correspondance il faut que le proces-

sus puisse accéder aux composants de l'application qu'il doit modifier. Les étapes de correspondance et de composition utilisent alors le modèle des composants concrets (cf. chapitre 8). Dans les différentes étapes du processus le configurateur peut avoir à intervenir pour résoudre des conflits (cf. figure 5.9). Nous donnons ici une description informelle du processus d'intégration de services, il est détaillé et formalisé dans les chapitres 7 et 8.

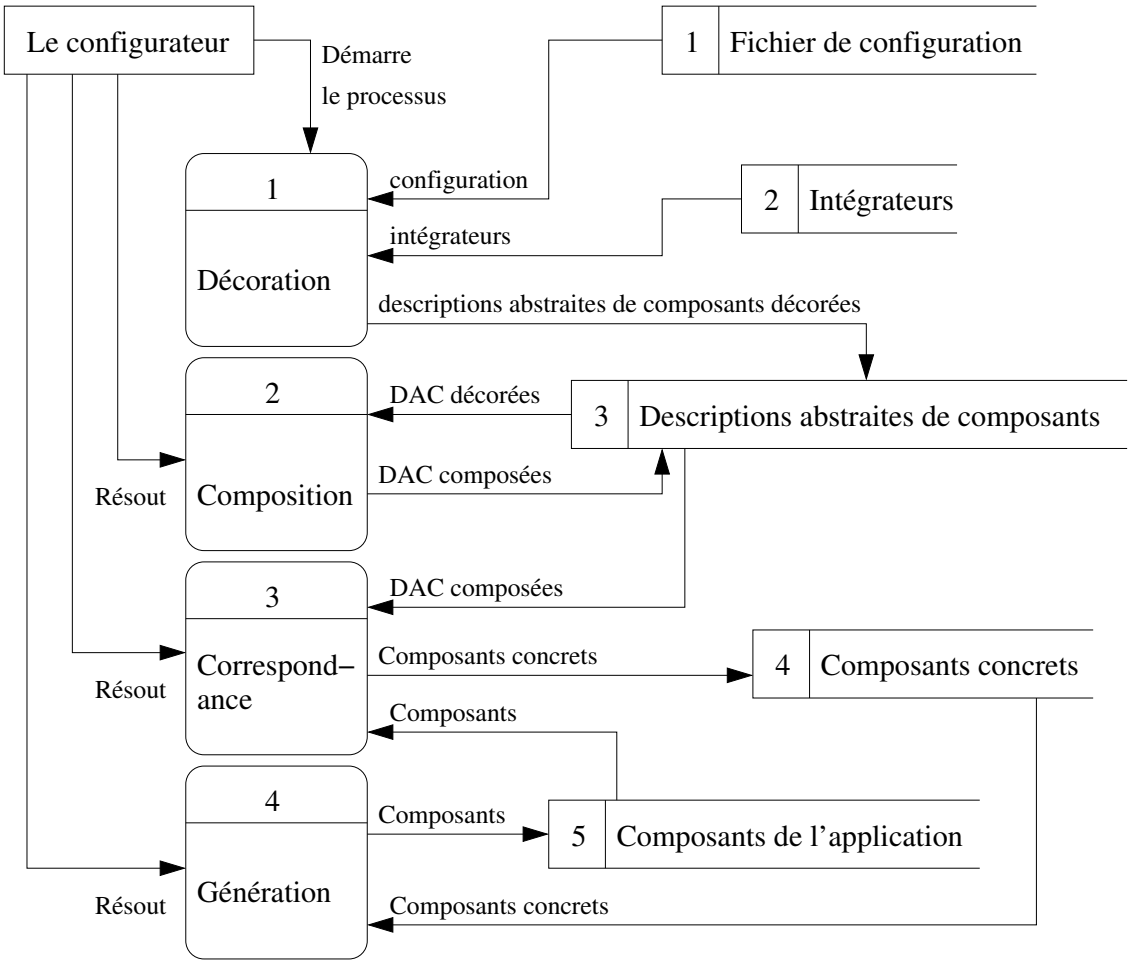

FIG. 5.9 – Flot de données dans le processus d'intégration de services

Comme nous l'avons vu précédemment notre modèle d'intégration de services est indépendant des systèmes de typage, visibilités, propriétés et règles ainsi que du plan de base pour l'évolution comportementale. Il en va de même pour les étapes du processus d'intégration de services qui sont décrites indépendamment de ces systèmes. Ici, comme pour les exemples d'intégrateurs précédents, nous utilisons les mêmes systèmes que dans la partie III.

5.3.1 Décoration

La première phase du processus d'intégration de services consiste à construire les représentations abstraites des composants à partir uniquement du fichier de configuration et des intégrateurs correspondants. Les descriptions abstraites de composants

ne contiennent au départ que le nom et une référence vers le composant ainsi que les types associés au composant. À partir du fichier de configuration nous ajoutons aux différentes descriptions abstraites de composants l'ensemble des évolutions structurelles et comportementales qui leurs sont attribuées par les intégrateurs. Nous obtenons une collection de descriptions abstraites de composants décorées par l'ensemble des évolutions définies dans les intégrateurs. Dans cette phase aucune vérification de conflit n'est effectuée.

Exemple :

A partir du fichier de configuration (cf. figure 5.8) et des intégrateurs (cf. figures 5.3, 5.4 et 5.5), on obtient, après décoration, la description abstraite de composant suivante pour le composant `MonComposantTest` (cf. figure 5.10).

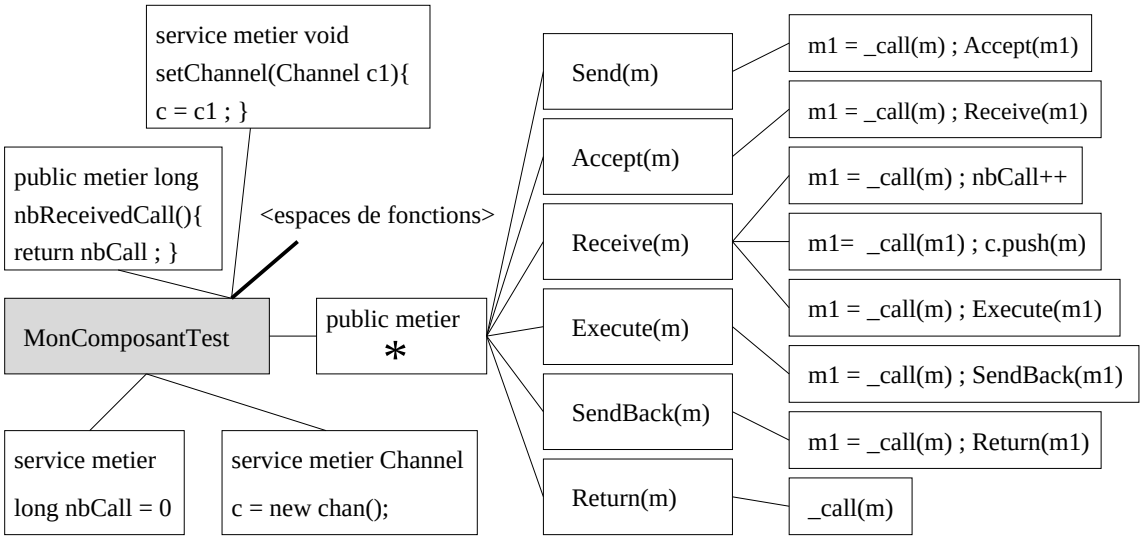

FIG. 5.10 – Création et décoration d'une description abstraite de composant

5.3.2 Composition

La seconde phase va composer pour chaque description abstraite de composant l'ensemble des évolutions structurelles et comportementales. Cette opération de composition est stable (il n'y a pas de perte d'information par rapport aux données) et elle fournit un résultat déterministe (le résultat est équivalent quelque soit l'ordre de déclaration des intégrateurs). De plus elle détecte les conflits entre les évolutions d'une même description abstraite de composant.

Dans un premier temps nous fusionnons les évolutions structurelles. Le processus détecte les conflits dus à une surcharge et peut opérer des fusions si nécessaire. Cette détection des conflits dépend des systèmes de typage, visibilité et propriétés qui font une comparaison entre les différentes évolutions (par exemple en CORBA la surcharge de nom de méthodes publiques est interdite). Le configurateur fait alors des choix pour résoudre ces conflits. Ensuite les évolutions comportementales sont fusionnées. Là aussi des conflits peuvent surgir qui dépendent du système de règles.

A la fin de cette phase, nous obtenons une collection de descriptions abstraites de composants dont les évolutions ont été fusionnées. Ces descriptions abstraites sont

toujours indépendantes des composants sur lesquels elles vont être appliquées. Cette phase de composition permet de détecter si des descriptions d'intégration de services sont conceptuellement incompatibles.

Exemple :

Dans le cas de la description abstraite de composant de `MonComposantTest`, il n'y a pas de conflit dû aux évolutions structurelles et on compose les trois évolutions comportementales qui portent sur le métaobjet `Receive`. Le résultat est donné figure 5.11. Par concision nous ne traitons que la signature générique `'public metier *'`.

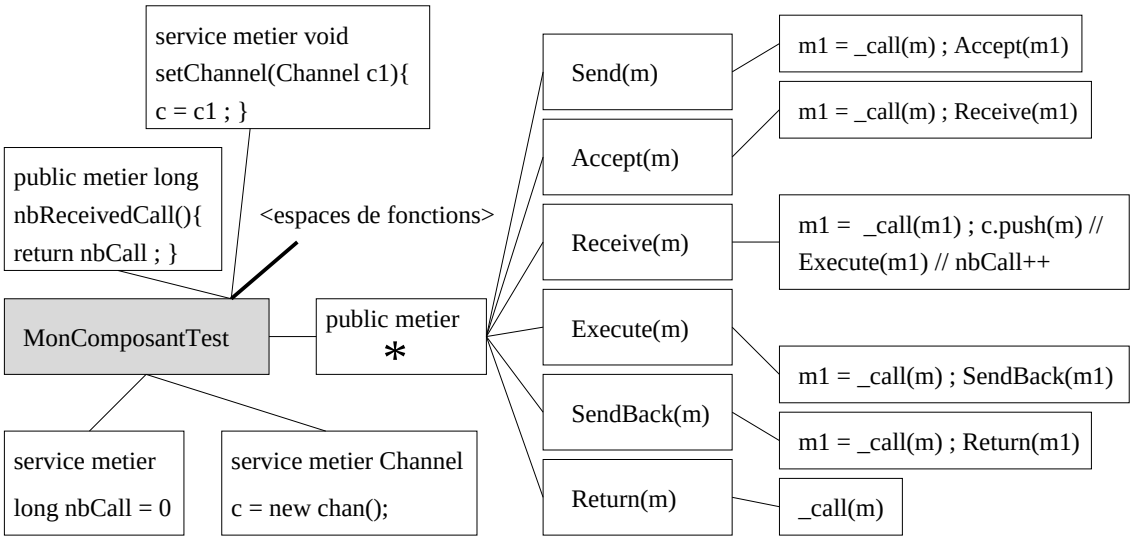

FIG. 5.11 – Composition des évolutions d'une description abstraite de composants

5.3.3 Correspondance

La troisième phase consiste à remonter les informations de l'application dans le modèle des composants concrets et à y transférer les évolutions des descriptions abstraites de composants. Ce modèle de composants concrets sert d'intermédiaire entre les descriptions abstraites de composants et les composants de l'application. L'objectif est de composer les informations du composant concret avec les informations de la description abstraite de composant.

Les méthodes et les attributs sont extraits du code des composants de l'application et introduits dans le composant concret à l'aide des informations de visibilité. Ces informations de visibilité sont issues du système de visibilité et de l'analyse du code essentiellement basée sur des visibilités liées à la cible et à l'architecture de la plateforme. Ensuite les évolutions des descriptions abstraites de composants sont ajoutées à ces composants concrets. Dans un premier temps ce sont les évolutions structurelles. Des conflits de nommage peuvent être détectés. Puis on ajoute aux signatures de méthodes les évolutions comportementales qui leur correspondent. Cette correspondance s'effectue entre les signatures des méthodes et les espaces de fonctions (qui sont des signatures génériques écrites à l'aide d'expressions régulières dans les intégrateurs).

Exemple :
La phase de correspondance avec le composant `MonTestComposant`, qui possède une méthode `compute` et une méthode `printString`, est représentée sur la figure 5.12 qui montre le remplacement, dans le composant concret, de l'espace de fonctions `'public metier *'` par les signatures de méthodes de `MonTestComposant`. La figure ne montre que la décoration de la signature de méthode `'int compute()'` pour rester concis.

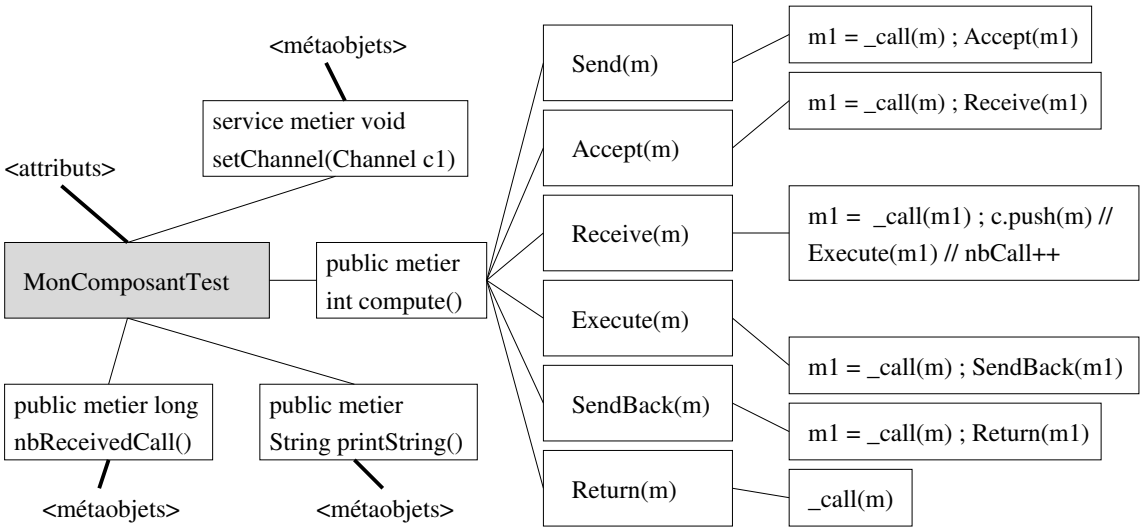

FIG. 5.12 – Composant concret qui représente la correspondance entre une description abstraite de composant et un composant cible.

A partir de cette phase on se sert d'une représentation des objets impliqués dans le fonctionnement du composant dans la plateforme cible. On nomme ces objets : *les objets d'implantation*.

Exemple :
Dans Jonas les objets impliqués pour faire fonctionner le composant sont l'interface, le proxy, le squelette, l'objet d'interposition, le bean, l'interface de la home, le proxy de la home, le squelette de la home, la home.

Ensuite pour chaque propriété (attribut ou méthode) et suivant sa visibilité on la décore par les objets d'implantation qui vont être impactés par ses évolutions. Une correspondance est établie entre les objets d'implantation présents sur une méthode et les métaobjets. On place alors les métaobjets sur les objets d'implantation correspondants.

Exemple :
Les objets d'implantation du composant concret impactés, dans le cas de la méthode `'int compute()'` qui possède une visibilité `public metier` sont l'interface, le proxy, le squelette, l'objet d'interposition et le bean. La figure 5.13 représente cette transformation sur la signature de méthode `'int compute()'`.

Enfin une phase de substitution dans les règles d'évolutions permet à l'aide d'une table de correspondance de substituer à chaque contrôle un appel effectif. On obtient

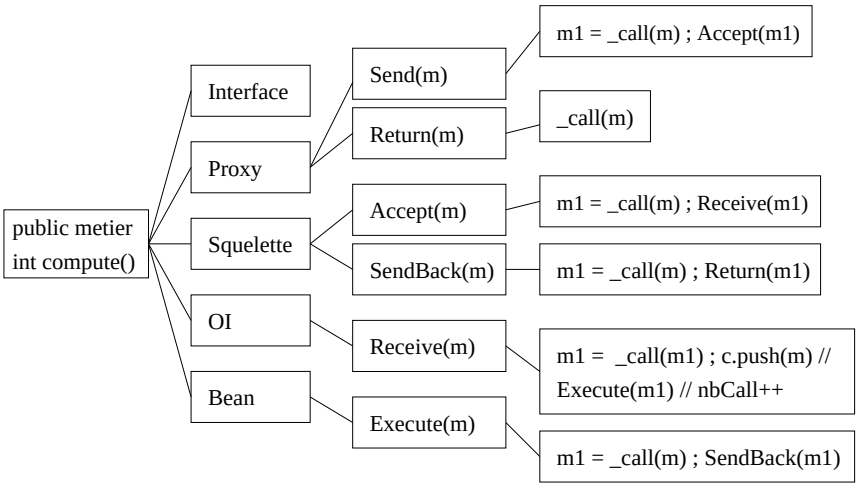

FIG. 5.13 – Objets d'implantation dans le composant concret

alors pour chaque méthode les appels effectifs à introduire dans les objets d'implantations.

`Exemple :`

La substitution des règles d'évolutions dans les objets d'implantation du composant concret donne la génération des appels effectifs suivants figure 5.14. Nous ne montrons que le cas de la méthode `'int compute()'` pour être concis.

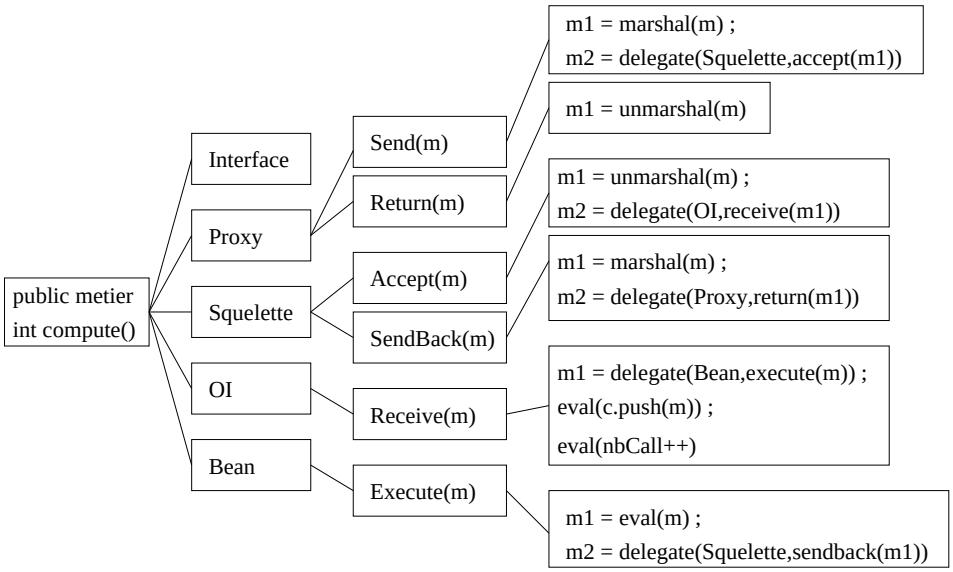

FIG. 5.14 – Substitution par des appels effectifs dans le composant concret

5.3.4 Génération

Dans cette dernière phase le processus d'intégration de services génère le code final de l'application à partir des composants concrets qui portent pour chaque objet

d'implantation les appels effectifs.

Plusieurs cas sont possibles dans le processus de génération. Nous pouvons utiliser le code des objets d'implantation déjà généré par la plateforme, soit les générer nous même pendant cette phase de génération. Nous considérons le cas où nous utilisons le code des objets d'implantation déjà généré par la plateforme. Dans ce cas là, certains contrôles comme l'empaquetage et la délégation sont déjà câblés dans les objets d'implantation. Il ne faut donc pas les surcharger, mais les utiliser comme points de branchements.

Exemple :

Nous montrons la génération de code dans le cas d'un EJB implémenté dans *Jonas*. Nous nous servons des objets générés par GenIC. Nous montrons pour la méthode `'int compute()'`, qui fait partie du composant original, comment le code des appels effectifs correspond au code du composant. Pour rester concis nous omettons le code de gestion des exceptions. Nous ne montrons pas non plus la modification de l'interface à laquelle nous ajoutons la signature de la méthode.

Le Proxy correspond au fichier `JOnASMonComposantTestImplRemote_stub.java`. Ici pour les métaobjets `Send` et `Return` les branchements ont déjà été générés par GenIC. Dans le cas d'une méthode ajoutée et suivant sa visibilité il faut soit la générer directement soit faire appel aux outils de génération de la plateforme.

Les appels effectifs du métaobjet `Send` correspondent à du code dans le `Proxy` :

Send(m) :

```
m1 = marshal(m) ;
m2 = delegate(Squelette,accept(m1))
```

Proxy :

```
java.rmi.server.RemoteCall call =
 ref.newCall((java.rmi.server.RemoteObject)this,operations,4,interfaceHash);
this.invoke(call);
```

Les appels effectifs du métaobjet `Return` correspondent à du code dans le `Proxy` :

Return(m) :

```
m1 = unmarshal(m)
```

Proxy :

```
java.io.ObjectInput in = call.getInputStream();
$result = in.readInt();
return $result;
```

Le Squelette correspond au fichier `JOnASMonComposantTestImplRemote_skel.java`. Ici pour les métaobjets `Accept` et `SendBack` comme précédemment les branchements ont déjà été générés par GenIC.

Les appels effectifs du métaobjet `Accept` correspondent à du code dans le `Squelette` :

Accept(m) :

```
m1 = unmarshal(m) ;
m2 = delegate(OI,receive(m1))
```

Squelette :

```
this.inRequest(call);
int $result = server.compute();
```

Les appels effectifs du métaobjet `SendBack` correspondent à du code dans le `Squelette` :

SendBack(m) :

```
m1 = marshal(m) ;
m2 = delegate(Proxy,return(m1))
```

Squelette :

```
java.io.ObjectOutput out = this.outReply(call);
out.writeInt($result);
```

L'objet d'interposition correspond au fichier `JOnASMonComposantTestImplRemote.java`. Ici pour le métaobjet `Receive` les branchements ont déjà été générés par GenIC, mais il faut ajouter les deux appels `eval`.

Receive(m) :

```
m1 = delegate(Bean,execute(m)) ;
eval(c.push(m)) ;
eval(nbCall++)
```

OI :

```
eb.JOnASMonComposantTestImplBean b =
 (eb.JOnASMonComposantTestImplBean) bctx.getInstance();
result = b.compute();
c.push(m) ;
nbCall++ ;
```

Ici il n'y a pas de métaobjet qui gère la valeur de retour de cette méthode sur l'objet d'interposition, on laisse donc l'objet d'interposition faire un retour normal.

OI :

```
return result;
```

Le Bean correspond au fichier `JOnASMonComposantTestImplBean.java`. Ici pour le métaobjet `Execute` les branchements ont déjà été générés par GenIC. Le `delegate` se traduit par une instruction `return`.

Execute(m) :

```
m1 = eval(m) ;
m2 = delegate(Squelette, sendback(m1))
```

Bean :

```
public int  compute()  {
return 1;
}
```

Conclusion

Dans ce chapitre nous avons détaillé notre approche de l'intégration de services. Nous avons étudié le rôle du configurateur et du fournisseur de service et nous avons expliqué l'ensemble du processus de l'intégration de services à l'aide d'exemples simples.

Notre approche permet de décrire l'intégration de services de manière abstraite sans utiliser les éléments des plateformes à composants sous-jacentes, grâce aux descriptions abstraites de composants. D'une part la description de l'intégration de services est facilitée car nous utilisons des concepts de plus haut niveau, et d'autre part

les descriptions d'intégration de services sont réutilisables pour des composants dans différentes plateformes à composants. Les descriptions d'intégration de services sont décrites indépendamment les unes des autres et sont composées de manière automatique. Cela permet une séparation concrète dans le développement des services et une meilleure réutilisation. De plus le fournisseur de services n'a pas à se soucier de l'intégration de son service avec les autres services. L'utilisateur de services peut paramétrer l'intégration des services à l'aide du fichier de configuration. De plus l'utilisateur de services peut modifier les intégrateurs de services s'il souhaite faire des adaptations plus profondes que celles permises par les paramètres. En effet comme les intégrateurs reposent sur un modèle abstrait, l'expression de l'intégration de services est facilitée.

Dans les chapitres suivants nous formalisons le modèle de descriptions abstraites de composants, le modèle d'intégrateurs ainsi que le modèle de composants concrets.

CHAPITRE 6

Un modèle pour l'évolution structurelle et comportementale des composants

Nous décrivons dans ce chapitre notre modèle des descriptions abstraites de composants. Comme nous l'avons vu dans l'étude des plateformes à composants (cf. chapitre 3), le comportement des composants est éparpillé dans les différents objets qui forment le composant. Pour répondre au problème de l'hétérogénéité de ces configurations d'objets, notre approche se base sur une modélisation du comportement des composants qui est indépendante de leur structure. Notre modèle permet de décrire des évolutions structurelles d'une part, qui touchent aux propriétés du composant, et des évolutions comportementales d'autres parts, qui affectent le comportement des composants.

Notre modèle pour l'évolution des composants est une abstraction des plateformes à composants. Notre modèle est assez général pour décrire différentes plateformes à composants mais contient assez d'informations pour permettre la projection des descriptions d'intégration de services dans ces diverses plateformes. Pour rester le plus général possible notre modèle utilise des systèmes externes pour le typage, la visibilité, les propriétés et les règles mais leur impose de respecter certaines contraintes. Nous montrons dans le chapitre 9 un choix possible pour ces systèmes qui couvrent les plateformes EJB, CCM et Fractal que nous avons étudiées dans la bibliographie.

Nous commençons par exposer les objets du modèle qui représentent les évolutions structurelles et comportementales des composants. Puis nous décrivons et formalisons les opérations sur le modèle. Enfin nous détaillons le modèle des intégrateurs et la phase de décoration.

6.1 Description abstraite de composant

L'élément principal de notre modèle est la *description abstraite de composant* (DAC) qui est une représentation d'un composant dans une plateforme à composant. Comme nous l'avons vu dans la section 5.3.1 de notre scénario de l'intégration de services, les descriptions abstraites de composants sont construites lors de la lecture du fichier de configuration.

Une description abstraite de composant est caractérisée par un nom, des attributs, des méthodes, des espaces de fonctions et des types. Le nom de la description abs-

traite de composant est un identifiant qui nous permet de référencer un composant lors de la lecture du fichier de configuration puis lors des phases de correspondance et de projection. Les attributs correspondent à des données ajoutées et utilisées par les services. Les méthodes décrivent un comportement ajouté au composant. Celles-ci peuvent être uniquement utiles à la mise en œuvre d'un service ou être utilisées comme un nouveau concept. La description du corps des méthodes ainsi que l'initialisation des attributs dépendent du langage. Les espaces de fonctions ont la charge de modéliser l'exécution du composant. Ils représentent des requêtes génériques sur lesquelles reposent les évolutions du comportement. Les types sont issus du modèle des intégrateurs (cf. section 6.5), ils vont permettre lors de la phase de correspondance (cf. chapitre 8) de vérifier qu'un composant respecte bien les interfaces attendues.

Il est important de noter qu'une description abstraite de composant représente un composant ainsi que son modèle d'exécution. La description abstraite de composant représente donc l'ensemble : composant et modèle d'exécution. Une description abstraite de composant est un type dans notre modèle, c'est donc une sous-classe de la classe `Type`.

Le type `DAC` (Description Abstraite de Composant) s'écrit ainsi :

DAC :

nom :	$tq\ nom \in Ident$
attributs :	$tq\ attributs := \{a_1, ..., a_n \mid a_i \in Attribut\}$
méthodes :	$tq\ méthodes := \{m_1, ..., m_n \mid m_i \in Méthode\}$
espacesFonctions :	$tq\ espacesFonctions := \{ef_1, ..., ef_n \mid ef_i \in EspaceDeFonctions\}$
types :	$tq\ types := \{t_1, ..., t_n \mid t_i \in Type\}$

Exemple :

De manière informelle voici la description abstraite de composant de `MonComposantTest` issue de la figure 5.11 :

$unDAC$:

nom :	$MonComposantTest$
attributs :	$service\ metier\ long\ nbCall\ =\ 0$ $service\ metier\ Channel\ c\ =\ new\ chan()$
méthodes :	$service\ metier\ void\ setChannel(Channel\ c1)\{c\ =\ c1\}$ $public\ metier\ long\ nbReceivedCall()\{return\ nbCall\}$
espacesFonctions :	$public\ metier\ *$: $-Send(m)$: $n := \_call(m); Accept(n)$ $-Accept(n)$: $m := \_call(n); Receive(m)$ $-Receive(m)$: $m1 := \_call(m); c.push(m)//$ $Execute(m1)//nbCall++$ $-Execute(m)$: $m1 := \_call(m); SendBack(m1)$ $-SendBack(m)$: $n := \_call(m); Return(n)$ $-Return(n)$: $\_call(n)$
types :	$\emptyset$

La vérification de l'intégration de services suppose une capacité à comparer les différents éléments qui représentent les évolutions à apporter aux composants. Les comparaisons diffèrent en fonction de la nature des éléments. Dans un souci de concision, nous noterons par l'opération `match` la vérification de l'*équivalence* entre deux éléments, et utilisons quelques fois $\equiv$ dans nos démonstrations pour simplifier l'écriture.

Pour définir les éléments constitutifs des descriptions abstraites de composants nous devons d'abord définir le système de typage. Le système du typage est un système externe au modèle. La première contrainte imposée au système de typage, due à l'étape de correspondance (cf. chapitre 8), est de définir une méthode `match` qui renvoie vrai si deux types sont équivalents. La seconde contrainte imposée au système de typage, due à l'étape de composition (cf. chapitre 7), est qu'il doit répondre à l'opération d'intersection (notée $\cap$). Cette opération renvoie un type qui est l'intersection de deux types ou le type $\perp$ qui représente le type vide s'il n'en existe pas. L'opération $\cap$ entre deux types doit avoir les propriétés suivantes :

$$\begin{array}{l} \forall t_1, t_2 \in Type \text{ alors } t_1 \cap t_2 \in Type. \\ \exists \perp \in Type \text{ tel que } t_1 \cap \perp \equiv \perp . \\ t_1 \equiv t_2 \Leftrightarrow t_1 \cap t_2 \neq \perp \\ \text{L'opération } \cap \text{ est commutative : } t_1 \cap t_2 \equiv t_2 \cap t_1. \\ \text{L'opération } \cap \text{ est transitive : } (t_1 \cap t_2) \cap t_3 \equiv t_1 \cap (t_2 \cap t_3). \end{array} \tag{6.1}$$

Les `attributs`, `méthodes`, `espacesFonctions` et `types` sont des collections de type `Collection` (respectivement de type `Attribut`, `Méthode`, `EspaceDeFonctions` et `Type`). La figure 6.1 représente le diagramme de classe UML de la description abstraite de composant.

Le type `Collection` (cf. figure 6.2) supporte une opération d'ajout d'un élément (la méthode `add`). Pour ajouter tous les éléments d'une collection existante nous utilisons la méthode `concat`. La méthode `contains` permet de savoir si un élément équivalent à celui passé en paramètre est présent dans la collection. La méthode `contains` renvoie vrai si un élément de la collection est équivalent (par la méthode `match`) à l'élément passé en paramètre ou faux sinon.

Le type `DAC` est complété par des opérations d'interrogation qui permettent de tester l'appartenance d'un attribut, d'une méthode, d'un espace de fonctions ou d'un type à une description abstraite de composant. Pour cela on fait une recherche en testant l'équivalence de chaque élément de la collection avec l'élément passé en paramètre. Nous définissons l'opération d'équivalence ci-après quand nous définissons ces différents types.

Appartenance d'un attribut à une description abstraite de composant :

$$attribut? : \left| \begin{array}{c} Attribut,\ DAC \rightarrow\ Boolean \\ \\ a, d \rightarrow attribut?(a, d) =\ d.attributs.contains(a) \end{array} \right.$$

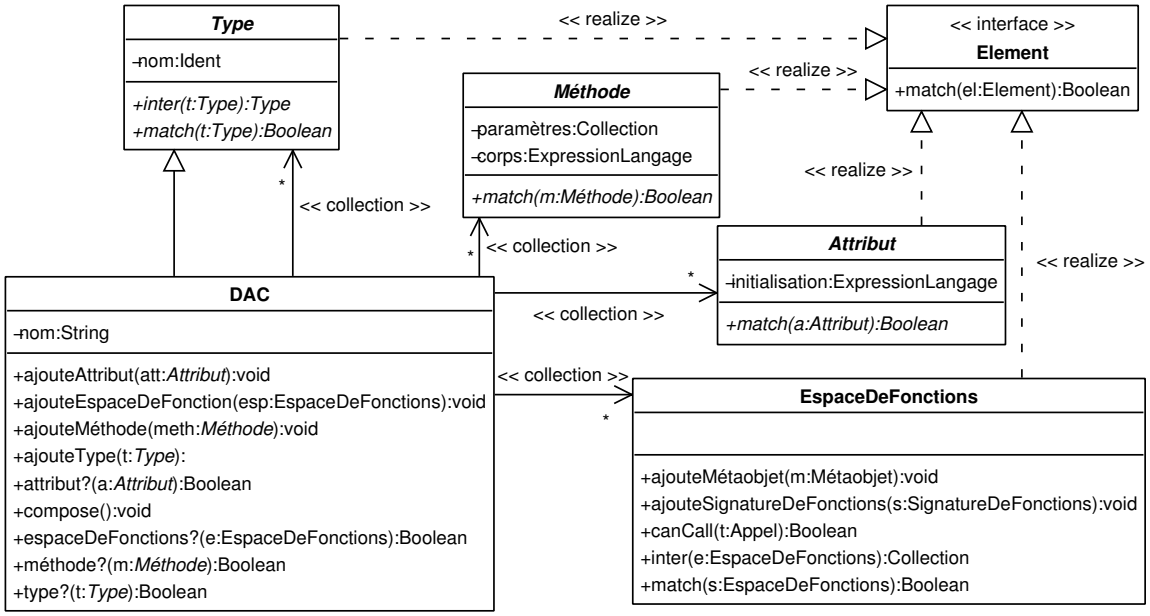

FIG. 6.1 – Description abstraite de composants

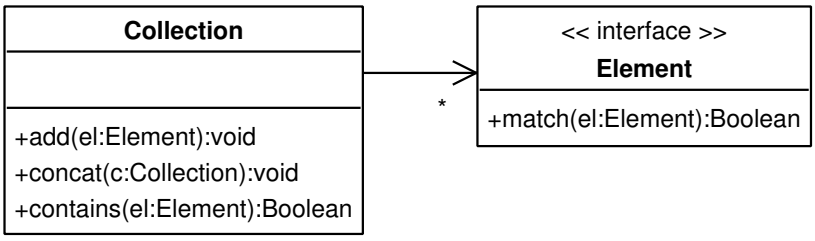

FIG. 6.2 – Le type Collection

Exemple :
Dans certains systèmes, deux attributs sont considérés comme équivalents s'ils ont le même nom et des types équivalents. Dans d'autres l'égalité sur le nom suffit à les considérer comme équivalents.

Appartenance d'une méthode à une description abstraite de composant :

$$méthode? : \left| \begin{array}{c} Méthode,\ DAC \rightarrow\ Boolean \\ \\ m, d \rightarrow méthode?(m, d) =\ d.méthodes.contains(m) \end{array} \right.$$

Exemple :
La différence sur les types de retour est le plus souvent omise dans la surcharge et deux méthodes sont le plus souvent considérées comme équivalentes si elles ont simplement le même nom et des signatures d'appels comparables.

Appartenance d'un espace de fonctions à une description abstraite de composant (la fonction d'équivalence sur deux espaces de fonctions est définie dans la section 6.3) :

$$espaceDeFonctions? : \left| \begin{array}{c} EspaceDeFonctions,\ DAC \rightarrow\ Boolean \\ \\ e, d \rightarrow espaceDeFonctions?(e, d) \\ = \\ d.espacesDeFonctions.contains(e) \end{array} \right.$$

Appartenance d'un type à une description abstraite de composant :

$$type? : \left| \begin{array}{c} Type,\ DAC \rightarrow\ Boolean \\ \\ t, d \rightarrow type?(t, d) =\ d.types.contains(t) \end{array} \right.$$

Maintenant que nous avons défini la description abstraite de composants et le système externe de typage, nous allons définir les types `Attributs`, `Méthodes` et `EspaceDeFonctions`.

6.2 Un modèle d'évolution structurelle

La partie qui supporte l'évolution structurelle de notre modèle est composé des attributs et des méthodes de la description abstraite de composant. Nous factorisons les éléments communs entre `Attribut` et `Méthode` dans `Propriété` (fig. 6.3). Le système de `Propriété` est un système externe au modèle de description abstraite de composant. Nous détaillons l'architecture minimum de ce système.

Une propriété se définit par une visibilité, un type et un nom. La visibilité détermine l'accès et le rôle de la propriété. La `Visibilité` est abstraite nous la considérons comme un système externe à notre modèle. La visibilité doit répondre aux opérations d'équivalence et d'intersection. Ces opérations doivent être commutatives pour l'étape de composition.

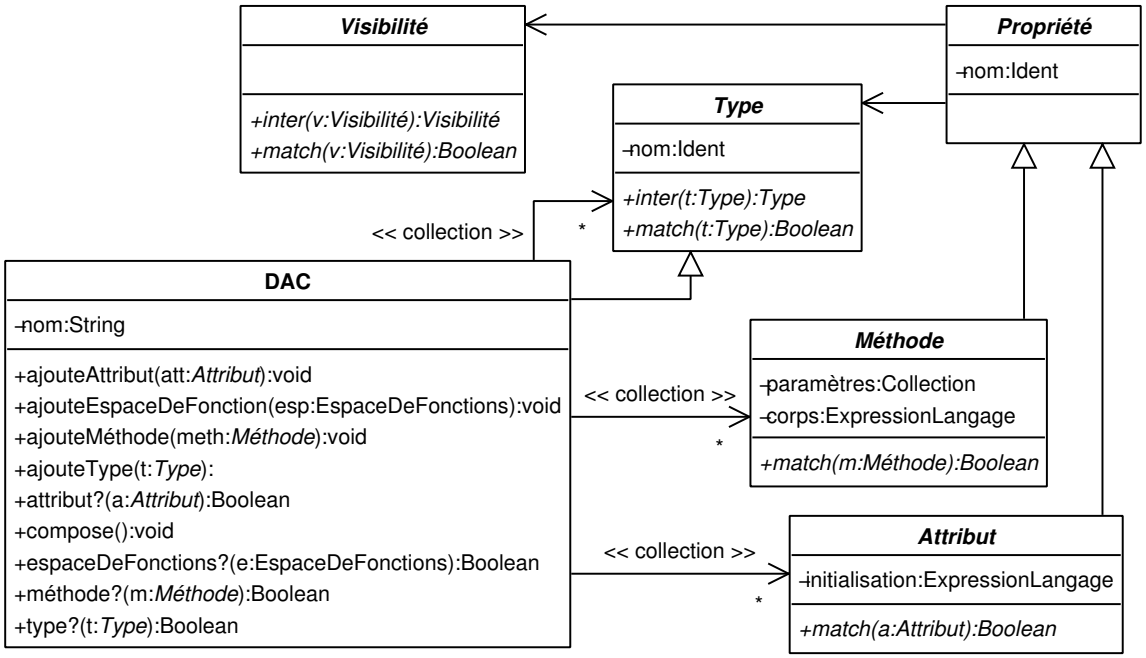

FIG. 6.3 – Support pour l'évolution structurelle

Exemple :
Les plateformes à composant ont besoin par exemple d'exprimer si une propriété est accessible depuis l'extérieur du composant (on dit alors qu'elle est `public`) ou bien si elle n'est accessible que par les services de la plateforme (on dit alors qu'elle est `service`).

Une propriété se décrit ainsi :

$$Propriété : \left| \begin{array}{ll} \text{visibilité :} & tq\ visibilité \in Visibilité \\ \text{type :} & tq\ type \in Type \\ \text{nom :} & tq\ nom \in Ident \end{array} \right.$$

6.2.1 Les attributs

Le type `Attribut` fait partie du système externe de propriété. Il est constitué, en plus de ce qu'il hérite du type `Propriété`, d'une initialisation qui est une expression du langage cible de type `ExpressionLangage`. Ainsi :

$$Attribut : \left| \begin{array}{ll} \text{initialisation :} & tq\ initialisation \in ExpressionLangage \end{array} \right.$$

L'opération d'équivalence entre deux attributs permet lors de la phase de composition (cf. chapitre 7) de vérifier si un attribut peut être en conflit avec un autre attribut. Cette opération n'est pas définie par le modèle mais laissée au système externe de propriété. En effet l'opération d'équivalence induit une dépendance trop forte envers le langage cible et la plateforme d'exécution.

6.2.2 Les méthodes

Le type `Méthode` fait aussi partie du système externe de propriété. Une méthode est décrite par une collection ordonnée de paramètres et un corps. La liste de paramètres est une collection ordonnée de couples `Type`, `Ident`. Le type `OrderedCollection` supporte les mêmes opérations que le type `Collection`. Et le corps est une collection ordonnée d'expressions du langage cible de type `ExpressionLangage`.

$$Méthode : \left| \begin{array}{ll} \text{paramètres :} & tq\ paramètres := \{t_1 : p_1, ..., t_n : p_n \mid \\ & (t_i \in\ Type) \wedge (p_i \in Ident)\} \\ \text{corps :} & tq\ corps := \{c_1, ..., c_n \mid c_i \in ExpressionLangage\} \end{array} \right.$$

Comme pour l'opération d'équivalence entre deux attributs, l'opération d'équivalence entre deux méthodes permet de vérifier si une méthode peut être en conflit avec une autre méthode et permet éventuellement de calculer l'intersection. Elle n'est pas définie par le modèle mais laissée au système externe de propriété.

6.3 Un modèle d'évolution comportementale

La partie qui supporte l'évolution comportementale correspond à la modélisation du flot des requêtes adressées à un composant. Dans la description abstraite de composant nous avons noté qu'un espace de fonctions représente des noms de requêtes génériques sur lesquelles nous posons les évolutions du comportement.

La diagramme de classe 6.4 présente la partie qui supporte l'évolution comportementale de notre modèle.

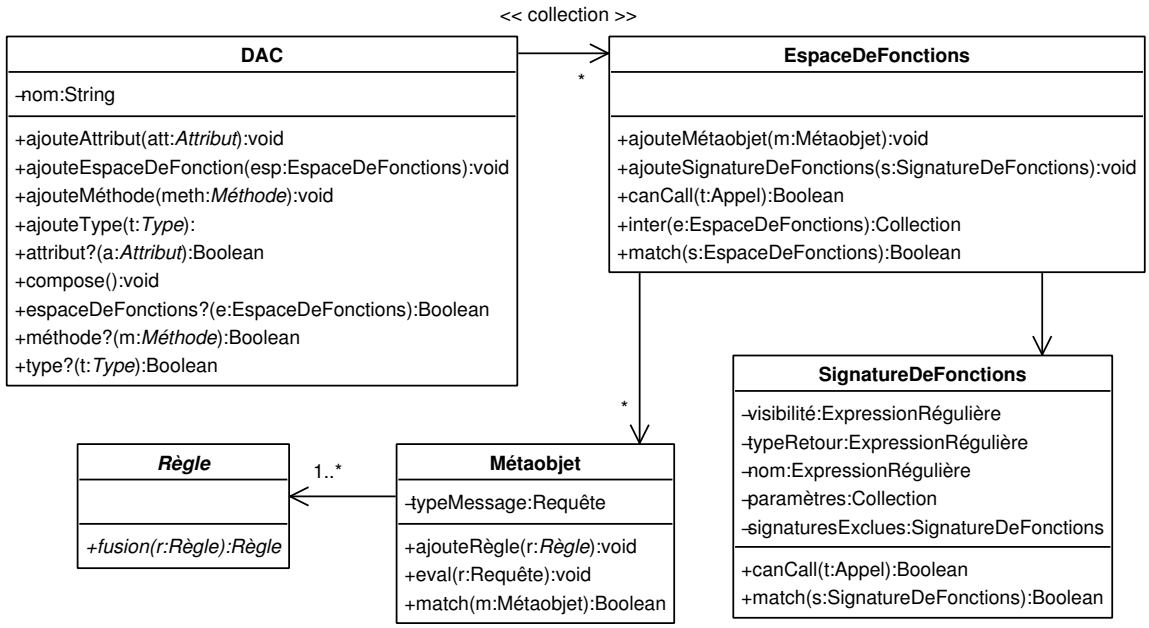

FIG. 6.4 – Métamodèle comportemental

Un élément de type `EspaceDeFonctions` est défini par une signature de fonctions générique et une collection de métaobjets.

$$EspaceDeFonctions : \left| \begin{array}{ll} \text{signatureDeFonctions :} & tq\ signatureDeFonctions \in SignatureDeFonctions \\ \text{métaobjets :} & tq\ métaobjets := \{m_1, ..., m_n \mid m_i \in Métaobjet\} \end{array} \right.$$

L'opération d'équivalence entre deux `EspaceDeFonctions` permet lors de la phase de décoration (cf. section 6.5.2) de vérifier si un autre espace de fonctions strictement équivalent au niveau des signatures de fonctions (par l'opération `match` des signatures de fonctions définie plus loin) existe. Dans ce cas plutôt que de créer un nouvel espace de fonction avec la même signature de fonctions on ajoute les métaobjets sur l'espace de fonctions déjà existant.

$$match : \left| \begin{array}{c} EspaceDeFonctions,\ EspaceDeFonctions \rightarrow\ Boolean \\ \\ e_1, e_2 \rightarrow match(e_1, e_2) \\ = \\ match(e_1.signatureDeFonctions, e_2.signatureDeFonctions) \end{array} \right.$$

D'autre part nous avons besoin d'une opération qui ne se base pas uniquement sur l'équivalence stricte mais prenne en compte le typage. Cette opération (nommée `canCall`) permet de savoir si un appel (de type `Appel`) est valide sur un espace de fonctions. L'opération `canCall` se base sur l'opération `canCall` des signatures de fonctions définie ci-après.

$$canCall : \left| \begin{array}{c} Appel,\ EspaceDeFonctions \rightarrow\ Boolean \\ \\ t, e \rightarrow canCall(t, e) = canCall(t, e.signatureDeFonctions) \end{array} \right.$$

Une `SignatureDeFonctions` supporte les expressions régulières ce qui permet de désigner des signatures de fonctions de manière générique. Une `SignatureDeFonctions` est décrite par une visibilité, un type de retour, un nom, une collection de types de paramètres et une collection de signatures exclues. L'attribut `signaturesExclues` permet de limiter l'espace couvert par la signature de fonctions. Cette collection d'exclusions est uniquement peuplée lors de la phase de composition (cf. chapitre 7), elle n'est pas directement accessible à la définition des intégrateurs.

Par construction, au moment de la fusion des éléments du modèle (cf. chapitre 7), certains cas produisent une signature de fonctions qui est vide. Nous noterons $\emptyset$ la signature vide.

$$
SignatureDeFonctions : \left|
\begin{array}{ll}
\text{visibilité :} & tq\ visibilité \in ExpressionRégulière \\
\text{typeRetour :} & tq\ typeRetour \in ExpressionRégulière \\
\text{nom :} & tq\ nom \in ExpressionRégulière \\
\text{typesParamètres :} & tq\ typesParamètres\ = \{t_1, ..., t_n \mid \\
 & (1 < i < n-1, t_i\ \in\ Type) \\
 & \&(t_n\ \in\ Type \cup \{'..'\})\ \} \\
\text{signaturesExclues :} & tq\ signaturesExclus\ = \{se_1, ..., se_n \mid \\
 & se_i\ \in\ SignatureDeFonctions\}
\end{array}
\right.
$$

Exemple :

Par exemple pour désigner les méthodes dont le nom commence par `set` qui sont accessibles de l'extérieur, renvoient `void`, font partie de la logique métier du composant et dont le nombre de paramètres est indifférent, alors la signature de fonctions s'écrit :

$$
uneSignatureDeFonctions : \left|
\begin{array}{ll}
\text{visibilité :} & public\ métier \\
\text{typeRetour :} & void \\
\text{nom :} & set* \\
\text{typesParamètres :} & \{..\} \\
\text{signaturesExclues :} & \emptyset
\end{array}
\right.
$$

L'opération d'équivalence entre deux `SignatureDeFonctions` est utilisée par l'opération `match` des espaces de fonctions. Elle est utilisé lors de la phase de décoration ce qui implique que la collection des signatures exclues est vide (elle n'est utilisée qu'à partir de la phase de composition), on ne fait donc pas de comparaison sur cet élément. Elle utilise une opération d'équivalence entre deux expressions régulières. Soient $er_1, er_2\ \in\ ExpressionRégulière$ alors on note $match(er_1, er_2)$ cette opération. L'équivalence entre deux signatures de fonctions est donc définie comme suit :

$$
match : \left|
\begin{array}{l}
SignatureDeFonctions,\ SignatureDeFonctions \rightarrow\ Boolean \\
\\
s_1, s_2 \rightarrow\ \ match(s_1, s_2) = \\
\\
\qquad match(s_1.visibilité, s_2.visibilité) \\
\qquad \&\ match(s_1.typeRetour, s_2.typeRetour) \\
\qquad \&\ match(s_1.nom, s_2.nom) \\
\qquad \&\ s_1.typesParamètres.size == s_2.typesParamètres.size \\
\\
\qquad \&\ \forall i\ \text{tq}\ 0 < i < s_1.typesParamètres.size\ \text{alors} \\
\qquad match(s_1.typesParamètres[i], s_2.typesParamètres[i])
\end{array}
\right.
$$

L'opération `canCall` prend en compte le typage pour savoir si un appel (de type `Appel`) est valide sur la signature de fonctions. Un appel de type `Appel` se définit par un nom et une suite d'arguments. Comme une signature de fonctions est définie par des expressions régulières, nous vérifions si l'appel t est dans l'espace conforme des mots défini par la signature de fonctions s noté $\mathcal{L}(s)$.

$$
canCall : \left| \begin{array}{ll} & Appel,\ SignatureDeFonctions \rightarrow\ Boolean \\ t, s \rightarrow & canCall(t,s) = t \in \mathcal{L}(s) \equiv \\ & t.nom \in \mathcal{L}(s.nom) \\ & \& \ t.retour \in \mathcal{L}(s.retour) \\ & \& \ t.typesParamètres \in \mathcal{L}(s.typesParamètres) \\ & \& \ \forall\ se \in s.signaturesExclues \text{ alors } not(canCall(t, se)) \end{array} \right.
$$

Les métaobjets représentent les différentes étapes de transition d'une requête. L'envoi d'un message d'un composant à un autre composant est représenté par l'application de traitements par des métaobjets sur la réification de ce message. Les traitements sont des règles de réécriture appliquées au message.

Un métaobjet est composé d'un nom, d'un type de message et d'une collection de règles. Le type du message correspond au type concret du message que le métaobjet est capable de traiter. Les règles sont les traitements à appliquer au message.

$$
Métaobjet : \left| \begin{array}{ll} \text{nom :} & tq\ nom \in Ident \\ \text{typeMessage :} & tq\ typeMessage \in Requête \\ \text{règles :} & tq\ règles = \{r_1, ..., r_n \mid r_i \in Règle\} \end{array} \right.
$$

Le type `Requête` est la super-classe abstraite pour définir la structure du message que peut traiter le métaobjet.

`Exemple :`

Nous en avons vu deux exemples dans la section 5.1 du chapitre précédent, qui montrent des requêtes sous forme de messages entre métaobjets locaux, et des requêtes sous forme de paquets réseaux entre métaobjets distants.

L'équivalence entre deux métaobjets sert lors de la phase de décoration (cf. section 6.5.2) pour éviter des métaobjets doublons dans le même espace de fonctions. Elle se définit en comparant leurs noms et leurs types de requêtes. Ainsi :

$$
match : \left| \begin{array}{ll} & Métaobjet,\ Métaobjet \rightarrow\ Boolean \\ m_1, m_2 \rightarrow match(m_1, m_2) = & m1.nom == m2.nom \\ & \& \ m1.requête == m2.requête \end{array} \right.
$$

Le type `Règle` définit des règles de réécriture qui vont permettre de faire évoluer le comportement des descriptions abstraites de composants. Le modèle est indépendant du système externe de règles de réécriture. Il requiert néanmoins que le système de

règles de réécriture supporte une opération de fusion qui permettent de composer les règles (utilisée lors de la phase de composition). Cette opération de fusion doit être commutative et associative. Soient $r_1, r_2, r_3 \in Règle$, l'opérateur de fusion (aussi noté ψ) vérifie alors :

$$\psi(r_1, r_2) \in Règle \cup Conflit$$
$$\psi(r_1, r_2) \equiv \psi(r_2, r_1)$$
$$\psi(r_1, \psi(r_2, r_3)) \equiv \psi(\psi(r_1, r_2), r_3)$$

6.4 Les opérations de décoration du modèle

Les opérations de décoration du modèle permettent d'ajouter des éléments aux descriptions abstraites de composants. Les opérations `ajouteAttribut`, `ajouteMéthode` et `ajouteType` permettent d'ajouter respectivement un attribut, une méthode et un type à une description abstraite de composant. Elles sont considérées comme des opérations d'évolutions structurelles puisqu'elles modifient la représentation structurelle de la description abstraite de composant. Les opérations `ajouteEspaceDeFonctions`, `ajouteMétaobjet` et `ajouteRègle` permettent respectivement d'ajouter un espace de fonctions à une description abstraite de composant, d'ajouter un métaobjet à un espace de fonctions et d'ajouter une règle à un métaobjet. Elles sont considérées comme des opérations d'évolutions comportementales.

6.4.1 Décoration pour l'évolution structurelle des descriptions abstraites de composants

L'ajout d'un attribut dans une description abstraite de composant se fait par l'ajout de celui-ci dans l'ensemble des attributs de la représentation abstraite de composant. Aucune vérification n'est réalisée concernant la possibilité d'un éventuel doublon déjà présent dans la liste. La vérification est faite pendant l'étape de composition (cf. chapitre 7).

L'ajout d'un attribut s'écrit donc :

$$ajouteAttribut : \left| \begin{array}{c} Attribut\ ,\ DAC\ \rightarrow\ DAC \\ \\ a, d \rightarrow ajouteAttribut(a, d) = d.attributs.add(a) \end{array} \right.$$

Nous procédons de même pour l'ajout d'une méthode. La vérification d'un conflit est, comme précédemment, faite pendant l'étape de composition.

L'ajout d'une méthode s'écrit donc :

$$ajouteMéthode : \left| \begin{array}{c} Méthode\ ,\ DAC\ \rightarrow\ DAC \\ \\ m, d \rightarrow ajouteMéthode(m, d) = d.méthodes.add(m) \end{array} \right.$$

On ajoute le type à la collection des types que s'il n'est pas déjà présent. Ici à la différence des attributs et des méthodes ce n'est pas un conflit d'ajouter plusieurs fois le même type, c'est simplement une redondance que l'on élimine. On écrit donc :

$$ajouteType : \left| \begin{array}{c} Type\ ,\ DAC\ \rightarrow\ DAC \\ \\ t, d \rightarrow ajouteType(t, d) = \text{Si } not(d.type?(t)) \text{ alors } d.types.add(t) \end{array} \right.$$

6.4.2 Décoration pour l'évolution comportementale des descriptions abstraites de composants

Pour les ajouts d'espace de fonctions et de métaobjets nous vérifions qu'il n'existe pas un élément équivalent auquel cas nous ne faisons pas l'ajout et nous laissons en place l'élément déjà présent.

L'ajout d'un espace de fonctions dans une description abstraite de composant s'écrit comme suit :

$$ajouteEspaceDeFonctions : \left| \begin{array}{c} EspaceDeFonctions\ ,\ DAC\ \rightarrow\ DAC \\ \\ ef, d \rightarrow ajouteEspaceDeFonctions(ef, d) \\ = \\ \text{Si } not(d.espaceDeFonctions?(ef)) \\ \text{alors } d.espacesFonctions.add(ef) \end{array} \right.$$

L'ajout d'un métaobjet dans un espace de fonctions s'écrit comme suit :

$$ajouteMétaobjet : \left| \begin{array}{c} Métaobjet\ ,\ EspaceDeFonctions\ \rightarrow\ EspaceDeFonctions \\ \\ m, ef \rightarrow ajouteMétaobjet(m, ef) \\ = \\ \text{Si } not(ef.métaobjet?(m)) \text{ alors } ef.métaobjets.add(m) \end{array} \right.$$

Les règles sont ajoutées sans vérification, elles seront composées dans la phase de composition (cf. chapitre 7) et des conflits pourront être détectés. L'ajout d'une règle dans un métaobjet s'écrit comme suit :

$$ajouteRègle : \left| \begin{array}{c} Règle\ ,\ Métaobjet\ \rightarrow\ Métaobjet \\ \\ r, m \rightarrow ajouteRègle(r, m) \equiv m.règles.add(r) \end{array} \right.$$

Le diagramme de classe 6.5 résume l'ensemble des éléments du modèle d'évolution structurelle et comportementale de composants.

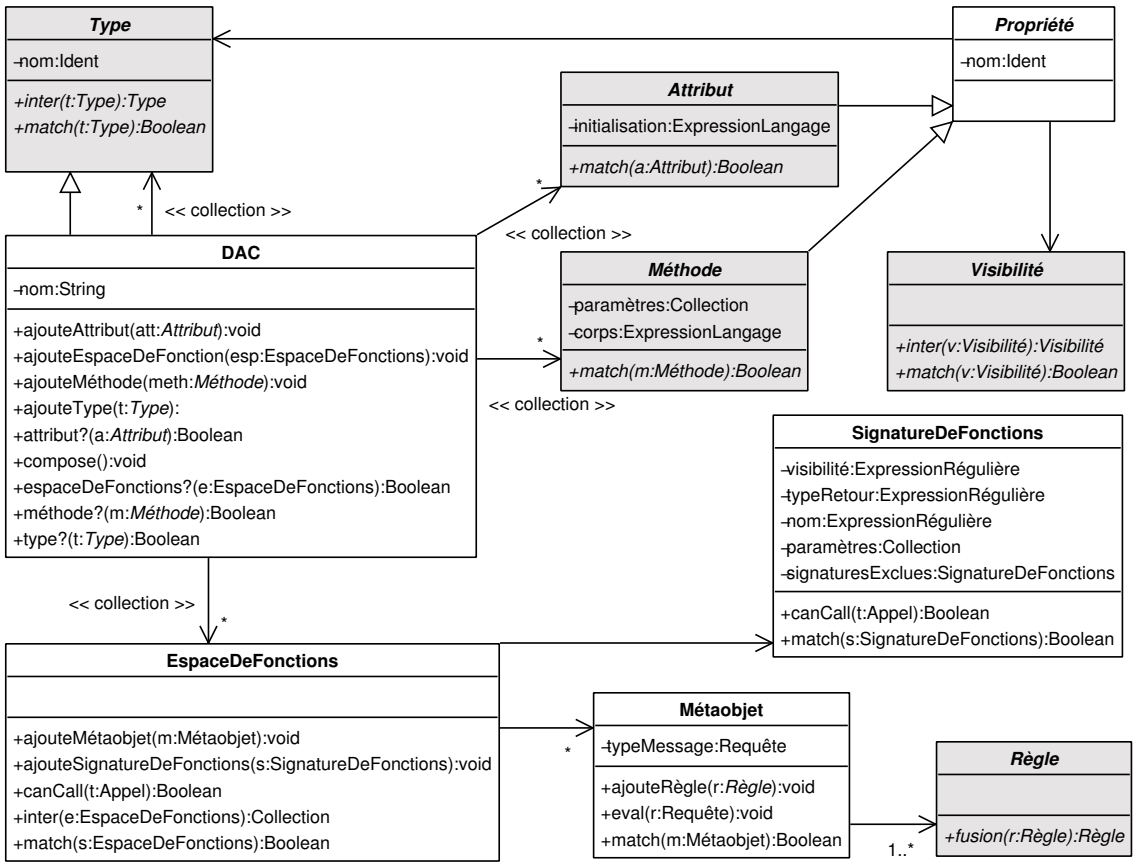

FIG. 6.5 – Résumé complet du modèle pour l'évolution structurelle et comportemental

6.5 Les intégrateurs : un modèle pour la description des intégrations de services

Pour faciliter l'utilisation de ce modèle de descriptions abstraites de composants, nous nous munissons d'un modèle d'*intégrateurs*. Ce modèle permet de décrire les intégrations de services qui sont alors transformées (par l'opération `apply()`) en appels d'opérations sur le modèle de descriptions abstraites de composants. Ce modèle d'intégrateur permet le paramétrage du modèle de description abstraite de composant. Un langage pour utiliser les intégrateurs est défini en annexe.

6.5.1 Le modèle des intégrateurs

L'élément principal de ce modèle est le type `Intégrateur` qui est composé d'une collection de paramètres, de type `Paramètre`, et d'une collection de règles d'intégration de type `RègleDIntégration`. On écrit alors :

$$Intégrateur : \left| \begin{array}{ll} \text{paramètres :} & tq\ paramètres = \{p_1, ..., p_n \mid p_i \in Paramètre\} \\ \text{règlesDIntégration :} & tq\ règlesDIntégration = \\ & \{ri_1, ..., ri_n \mid ri_i \in RègleDIntégration\} \end{array} \right.$$

Un paramètre est constitué d'un nom et d'un type. Les paramètres peuvent être soit des descriptions abstraites de composants, soit des types, soit des collections d'objets comme par exemple une collection de méthodes. On écrit :

$$Paramètre : \left| \begin{array}{ll} \text{nom :} & tq\ nom \in Ident \\ \text{type :} & tq\ type \in Type \end{array} \right.$$

Les règles d'intégration permettent de décrire les évolutions des composants. Toute règle d'intégration porte sur un paramètre qu'elle a en charge de modifier. Le type `RègleDIntégration` est le super-type qui va permettre de décrire d'une part les évolutions structurelles et d'autre part les évolutions comportementales. Ainsi :

$$RègleDIntégration : \left| \begin{array}{ll} \text{paramètre :} & tq\ paramètre \in Paramètre \end{array} \right.$$

Une règle d'évolution structurelle permet soit d'ajouter un attribut, soit d'ajouter une méthode. La définition d'ajout d'un attribut s'exprime directement par la déclaration d'un attribut. De même la définition d'ajout d'une méthode s'exprime directement par la déclaration d'une méthode.

Une règle comportementale est définie par une signature de fonctions, un métaobjet et une règle de réécriture. On écrit donc :

$$RègleComportementale : \left| \begin{array}{ll} \text{signatureDeFonctions :} & tq\ signatureDeFonctions \in \\ & SignatureDeFonctions \\ \text{métaobjet :} & tq\ métaobjet \in Métaobjet \\ \text{règle :} & tq\ règle \in Règle \end{array} \right.$$

Le schéma UML 6.6 résume le modèle d'intégrateur.

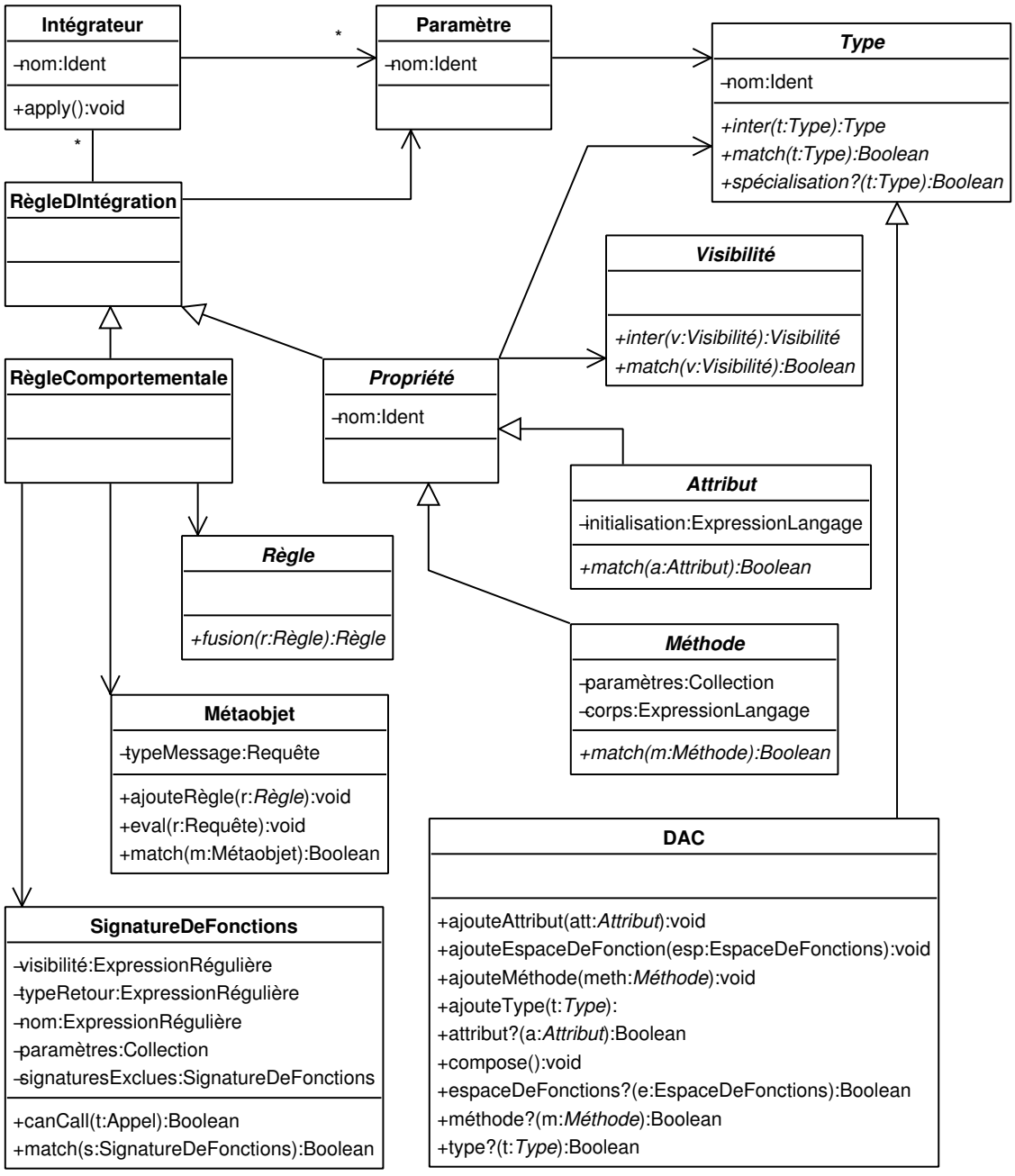

FIG. 6.6 – Modèle pour la description des intégrations de services

6.5.2 La phase de décoration par l'opération `apply()`

L'opération `apply()` sur un `Intégrateur` permet de décorer les descriptions abstraites de composants. Elle se base sur la lecture du fichier de configuration qui va permettre de substituer les noms de variables des intégrateurs par les valeurs du fichier de configuration.

Nous nous servons d'un `DACManager` pour gérer les descriptions abstraites de composants qui nous sert à récupérer une description abstraite de composant à partir de son nom. Le `DACManager` renvoie la description abstraite de composant de nom correspondant ou bien crée une nouvelle description abstraite de composant avec ce nom s'il n'en existait pas déjà une.

Nous évitons aussi la création de doublons pour les espaces de fonctions, pour cela nous vérifions qu'il n'existe pas déjà un espace de fonctions dont la signature est strictement équivalente (par la méthode `match`). De même pour les métaobjets nous évitons les doublons en vérifiant qu'il n'existe pas de métaobjets strictement équivalent (par la méthode `match`).

$$
\begin{array}{|l}
\multicolumn{1}{c}{Intégrateur \rightarrow void} \\
\\
i \rightarrow apply(i) = \quad \forall\, r \in i.règleDIntégration \text{ faire} \\
\qquad\qquad\qquad\qquad DAC\ d\ =\ DACManager.get(r.paramètre.nom); \\
\\
\qquad\qquad\qquad\qquad d.ajouteType(r.paramètre.type); \\
\\
\qquad\qquad\qquad\qquad \text{Si } r \in Attribut \text{ alors } d.ajouteAttribut(r); \\
apply : \\
\qquad\qquad\qquad\qquad \text{Si } r \in Méthode \text{ alors } d.ajouteMéthode(r); \\
\\
\qquad\qquad\qquad\qquad \text{Si } r \in RègleComportementale \text{ alors} \\
\qquad\qquad\qquad\qquad ef = EspaceDeFonctions(r.signatureDeFonctions); \\
\qquad\qquad\qquad\qquad d.ajouteEspaceDeFonctions(ef); \\
\qquad\qquad\qquad\qquad m = r.métaobjet; \\
\qquad\qquad\qquad\qquad ef.ajouteMétaobjet(m); \\
\qquad\qquad\qquad\qquad m.ajouteRègle(r.règle); \\
\end{array}
$$

Conclusion

Dans ce chapitre nous avons détaillé un modèle de descriptions abstraites de composants qui supporte l'évolution de ces composants. Nous avons typé l'ensemble des éléments et donné les opérations d'ajout d'évolutions du modèle. Notre modèle repose sur les systèmes externes de types, visibilités, propriétés et règles qui sont à définir pour l'utilisation du modèle.

Nous avons aussi décrit un modèle d'intégrateurs qui permet la manipulation du modèle de descriptions abstraites de composants. Ce modèle d'intégrateurs permet d'ajouter des évolutions à l'aide de règles d'intégration dans le modèle de descriptions abstraites de composants. Nous avons aussi montré au travers de l'opération `apply()` le processus de décoration des descriptions abstraites de composants.

Le chapitre suivant introduit l'opérateur de composition sur les descriptions abstraites de composants qui permet de détecter des conflits entre les différentes évolutions et de les fusionner. Cette opération de composition offre les propriétés d'unicité du résultat et de stabilité des éléments.

CHAPITRE 7

Fusion des éléments du modèle d'évolution

Dans ce chapitre nous étudions le processus de composition qui s'applique aux descriptions abstraites de composants. Les opérations de composition permettent d'une part de détecter des conflits entre les différentes évolutions ajoutées, et d'autre part de composer les évolutions qui sont appliquées sur les mêmes éléments. Dans cette phase les descriptions abstraites de composants sont toujours indépendantes des composants auxquels elles correspondent. Cette phase de composition permet de détecter si des descriptions d'intégration de services sont incompatibles, et par le biais de la résolution des conflits de les rendre compatibles.

Notre objectif est que les opérations de composition fournissent un résultat unique quelque soit l'ordre de traitement des évolutions, et que le résultat soit déterministe. De plus la composition doit assurer la stabilité et l'absence de perte d'éléments.

Dans ce chapitre nous définissons l'opérateur de composition (noté ρ) qui doit répondre aux propriétés suivantes quand aucun conflit n'est levé :

$d \rightarrow \rho(d) =$ $d_1 \in DAC \cup Conflit$ tq :

$\forall\ a_i, a_j \in d_1.attributs\ ,$ si $i \neq j$ alors $a_i \not\equiv a_j$
$\forall\ i,\ d_1.attributs.contains(d.attributs[i])$

$\forall\ m_i, m_j \in d_1.méthodes\ ,$ si $i \neq j$ alors $m_i \not\equiv m_j$
$\forall\ i,\ d_1.méthodes.contains(d.méthodes[i])$

$\forall\ e_i, e_j \in d_1.espacesDeFonctions\ ,$ si $i \neq j$ alors $e_i \cap e_j = \emptyset$
$\forall\ t \in Appel$ tq $t \in \mathcal{L}(d.espacesDeFonctions)$
alors $t \in \mathcal{L}(d_1.espacesDeFonctions)$

La composition des évolutions structurelles doit détecter des conflits de surcharge. Pour la stabilité nous devons vérifier que tous les attributs et toutes les méthodes de la description abstraite de composant non composée soient bien présents dans la description abstraite de composant une fois composée.

La composition des évolutions comportementales doit dans un premier temps partitionner l'ensemble des espaces de fonctions puis vérifier qu'il n'y a pas de conflits entre les métaobjets et finalement fusionner les règles s'il n'y pas de conflits. La stabilité doit vérifier que tous les appels que l'on peut faire sur la description abstraite

de composant avant la composition sont faisables sur la description abstraite de composant résultant de la composition.

Le résultat de l'opération ρ sur les descriptions abstraites de composants se calcule comme suit :

$$\rho : \left| \begin{array}{l} DAC \rightarrow DAC \cup Conflit \\ d \rightarrow \rho(d) = \quad d_1 \in DAC \text{ tq :} \\ \qquad\qquad d_1.attributs = \rho_{attributs}(d.attributs) \\ \qquad\qquad d_1.méthodes = \rho_{méthodes}(d.méthodes) \\ \qquad\qquad d_1.espacesDeFonctions = \rho_{espacesDeFonctions}(d.espacesDeFonctions) \end{array} \right.$$

L'opération ρ sur les descriptions abstraites de composants applique respectivement les opérations de composition $\rho_{attributs}$, $\rho_{méthodes}$ et $\rho_{espacesDeFonctions}$ à ses attributs `attributs`, `méthodes` et `espacesDeFonctions`.

Dans un premier temps nous donnons les définitions des opérateurs de composition. Puis nous faisons la démonstration de l'unicité du résultat de l'opérateur ρ sur les descriptions abstraites de composants ainsi que de la stabilité.

7.1 Composition des éléments de l'évolution structurelle

Pour la composition des évolutions structurelles il faut appliquer l'opérateur de composition $\rho_{attributs}$ sur les attributs et d'autre part $\rho_{méthodes}$ sur les méthodes. La composition des évolutions structurelles vérifie qu'il n'y a pas de conflit de surcharge.

La recherche de conflits sur l'ensemble des attributs se fait sur l'ensemble des attributs d'une représentation abstraite de composant en testant leur équivalence deux à deux à l'aide de la méthode `contains` qui utilise la méthode `match` (cf. chapitre 6). Si deux attributs sont équivalents alors il y a un conflit. Dans le cas d'un conflit on stocke les informations relatives aux conflits pour les faire resurgir à la fin du processus pour qu'ils puissent être traités. On écrit alors :

$$\rho_{attributs} : \left| \begin{array}{l} Collection < Attribut > \rightarrow Collection < Attribut > \cup Conflit \\ ca \rightarrow \rho_{attributs}(ca) = \quad ca_1 \in Collection < Attribut > \text{ tq :} \\ \qquad\qquad \forall i \text{ tq } 0 < i < ca.size \text{ alors} \\ \qquad\qquad \text{si } ca_1[i+1 : ca.size].contains(ca[i]) \\ \qquad\qquad \text{alors } new\ Conflit(ca[i]) \\ \qquad\qquad \text{sinon } ca_1.add(ca[i]) \end{array} \right.$$

De même la recherche de conflits sur l'ensemble des méthodes se fait en testant l'égalité de l'ensemble des méthodes d'une représentation abstraite de composant deux à deux. On écrit alors :

$$\rho_{méthode} : \left| \begin{array}{l} Collection < Méthode > \rightarrow\ Collection < Méthode > \cup Conflit \\ \\ cm \rightarrow \rho_{méthode}(ca) = \quad cm_1 \in Collection < Méthode >\ \textbf{tq} : \\ \qquad\qquad\qquad\qquad\quad \forall i\ \textbf{tq}\ 0 < i < cm.size\ \textbf{alors} \\ \qquad\qquad\qquad\qquad\quad \textbf{si}\ cm[i+1 : cm.size].contains(cm[i]) \\ \qquad\qquad\qquad\qquad\quad \textbf{alors}\ new\ Conflit(cm[i]) \\ \qquad\qquad\qquad\qquad\quad \textbf{sinon}\ cm_1.add(cm[i]) \end{array} \right.$$

S'il n'y a pas de conflit ces deux opérations sont stables. En effet tous les éléments de la collection initiale se retrouvent dans la collection résultante. Dans le cas où des conflits surviennent, ils doivent être résolus et l'opération est relancée.

7.2 Composition des éléments de l'évolution comportementale

La composition des évolutions comportementales se fait en appliquant l'opérateur $\rho_{espacesDeFonctions}$ sur les espaces de fonctions. Comme nous le détaillons ci-après cet opérateur repose sur les opérateurs $\rho_{signaturesDeFonctions}$ et $\rho_{métaobjets}$. Pour chacun de ces opérateurs nous allons montrer qu'il y a unicité du résultat et stabilité par rapport à l'ensemble de départ.

La première étape de la composition des évolutions comportementales est de disjoindre les espaces de fonctions pour pouvoir calculer pour chacun d'eux leur ensemble de métaobjets et de règles associées.

La composition d'une collection d'espaces de fonctions renvoie une collection d'espaces de fonctions équivalentes et partitionnées. Pour effectuer le calcul de la collection résultante nous nous servons d'une opération annexe qui calcule la partition. Cette opération prend en paramètres un espace de fonctions et une collection d'espaces de fonctions déjà partitionnée. Nous écrivons donc :

$$\rho_{espacesDeFonctions} : \left| \begin{array}{l} Collection < EspaceDeFonctions > \\ \rightarrow \\ Collection < EspaceDeFonctions > \\ \\ ce \rightarrow \quad \rho_{espacesDeFonctions}(ce) = \\ \qquad ce_1 \in Collection < EspaceDeFonctions >\ \textbf{tq} : \\ \qquad \forall i,\ 0 < i < ce.size\ \textbf{alors}\ ce_1.concat(\rho_{cedf}(ce[i], ce_1)) \end{array} \right.$$

L'opération de composition ρ_{cedf} prend en paramètres un espace de fonctions et une collection d'espaces de fonctions bien partitionnées. Cette opération renvoie alors une nouvelle collection d'espaces de fonctions bien partionnée :

$$\rho_{cedf} : \left| \begin{array}{l} EspaceDeFonctions\ ,\ Collection < EspaceDeFonctions > \\ \rightarrow \\ Collection < EspaceDeFonctions > \\ \\ ef, ce \rightarrow \rho_{cedf}(ef, ce) = \quad ce_1 \in Collection < EspaceDeFonctions > \text{ tq :} \\ \qquad \text{si } ce = \emptyset \text{ alors } ce_1 = \{ef\} \\ \qquad \text{sinon si } ce = \{e_1, ..., e_n\} \text{ alors} \\ \qquad ce_1 = \{e_1 \setminus ef,\ e_1 \cap ef,\ ...\ ,\ e_n \setminus ef,\ e_n \cap ef,\ ef \setminus ce\} \end{array} \right.$$

Pour pouvoir montrer que $\rho_{espacesDeFonctions}$ renvoie une solution unique et est stable, il nous faut d'abord définir les opérations ensemblistes $\cap$, $\setminus$ sur les espaces de fonctions et $\setminus$ entre un espace de fonctions et une collection d'espaces de fonctions.

7.2.1 Définition de l'opérateur $\cap$ entre deux espaces de fonctions

Soient $e_1, e_2 \in EspaceDeFonctions$ alors l'intersection $e_1 \cap e_2$ renvoie un espace de fonctions qui correspond à l'intersection des signatures de e_1 et e_2 et à la composition de leurs collections de métaobjets. L'opération $\rho_{métaobjets}$ peut émettre un conflit. Dans ce cas le conflit est remonté jusqu'au configurateur de l'application qui le résout et la configuration est mise à jour. On écrit alors :

$$\cap : \left| \begin{array}{l} EspaceDeFonctions,\ EspaceDeFonctions \rightarrow\ EspaceDeFonctions \cup Conflit \\ \\ e_1, e_2 \rightarrow e_1 \cap e_2 =\ e \in EspaceDeFonctions \text{ tq :} \\ e.espaceDeFonctions = e_1.signatureDeFonctions \cap e_2.signatureDeFonctions \\ e.métaobjets = \rho_{métaobjets}(e_1.métaobjets, e_2.métaobjets) \end{array} \right.$$

Il nous faut définir l'opération d'intersection sur les signatures de fonctions et l'opération de composition sur les collections de métaobjets.

7.2.1.1 Définition de l'opérateur $\cap$ entre deux métaobjets

La définition de l'opération $\rho_{métaobjets}$ sur deux collections de métaobjets renvoie une collection de métaobjets dans laquelle les métaobjets strictement équivalents (par la méthode match) ont leurs règles de réécritures fusionnées. Dans le cas où deux métaobjets de même nom utilisent des types de requêtes différents alors il y a un conflit. On écrit :

$$
\rho_{métaobjets} : \left|
\begin{array}{ll}
 & Collection < Métaobjet > , Collection < Métaobjet > \\
 & \rightarrow \\
 & Collection < Métaobjet > \cup Conflit \\
 & \\
cm_1, cm_2 \rightarrow & \rho_{métaobjets}(cm_1, cm_2) = \\
 & cm = new\ Collection < Métaobjet > \text{ et } \forall cm_1[i] \in cm_1 \text{ tq :} \\
 & \\
 & \text{Si } \exists\ cm_1[i] \in cm_1 \text{ et Si } \exists\ cm_2[j] \in cm_2 \text{ tq :} \\
 & cm_1[i].nom \equiv cm_2[j].nom \\
 & \&\ cm_1[i].typeMessage \equiv cm_2[j].typeMessage \\
 & \text{Alors } cm.add(m) \text{ tq :} \\
 & \quad Métaobjet\ m = new\ Métaobjet(); \\
 & \quad m.nom = cm_1[i].nom \\
 & \quad m.typeMessage = cm_1[i].typeMessage \\
 & \quad m.règles = \psi(cm_1[i].règles, cm_2[j].règles) \\
 & \\
 & \text{Si } \exists\ cm_1[i] \in cm_1 \text{ et Si } \exists\ cm_2[j] \in cm_2 \text{ tq :} \\
 & cm_1[i].nom \equiv cm_2[j].nom \\
 & \&\ cm_1[i].typeMessage \not\equiv cm_2[j].typeMessage \\
 & \text{Alors } new\ Conflit(cm_1[i]) \\
 & \\
 & \text{Sinon si} \exists\ cm_1[i] \in cm_1 \text{ et Si } \exists\ cm_2[j] \in cm_2 \text{ tq :} \\
 & cm_1[i].nom \not\equiv cm_2[j].nom \\
 & \&\ cm_1[i].typeMessage \not\equiv cm_2[j].typeMessage \\
 & \text{Alors } cm = cm_1[i]\ \&\ cm_2[j]
\end{array}
\right.
$$

Par cette définition de l'opérateur $\rho_{métaobjets}$ et du fait que le système de règles définisse une propriété de fusion commutative et associative (notée ψ), nous obtenons par construction l'unicité et la stabilité de l'opération $\rho_{métaobjets}$. Ainsi nous avons :

$$
\begin{array}{c}
\rho_{métaobjets}(cm_1, cm_2) \equiv \rho_{métaobjets}(cm_2, cm_1) \\
\rho_{métaobjets}(\rho_{métaobjets}(cm_1, cm_2), cm_3) \equiv \rho_{métaobjets}(cm_1, \rho_{métaobjets}(cm_2, cm_3))
\end{array}
\tag{7.1}
$$

7.2.1.2 Définition de l'opérateur ∩ entre deux signatures de fonctions

L'intersection de deux signatures de fonctions $s_1, s_2 \in SignatureDeFonctions$ retourne une signature de fonctions qui est une signature valide dans s_1 et s_2. On écrit alors :

$$\cap : \left| \begin{array}{l} SignatureDeFonctions\ ,\ SignatureDeFonctions \\ \rightarrow \\ SignatureDeFonctions \end{array} \right.$$

$$\begin{array}{ll} s_1, s_2 \rightarrow & s_1 \cap s_2 = s \in SignatureDeFonctions \text{ tq :} \\ & s.visibilité = s_1.visibilité \cap s_2.visibilité \\ & s.typeRetour = s_1.typeRetour \cap s_2.typeRetour \\ & s.nom = s_1.nom \cap s_2.nom \\ & s.typesParamètres = s_1.typesParamètres \cap s_2.typesParamètres \\ & s.signaturesExclues = s_1.signaturesExclues \cup s_2.signaturesExclues \end{array}$$

L'opération $\cup$ sur les collections de signatures exclues correspond à la concaténation des deux ensembles (par l'opération `concat`).

7.2.1.3 Définition de l'opérateur $\cap$ entre deux expressions régulières

L'opération $\cap$ sur la visibilité, le type de retour et le nom des espaces de fonctions se définit par rapport à l'opération d'intersection du langage des expressions régulières. Les expressions régulières sont issues de la théorie des automates et de la théorie des langages formels, nous prenons comme acquis les différentes démonstrations sur les expressions régulières de [36].

On écrit donc l'opération $\cap$ sur les expressions régulières comme suit :

$$\cap : \left| \begin{array}{l} ExpressionRégulière\ ,\ ExpressionRégulière\ \rightarrow\ ExpressionRégulière \\ \\ er_1, er_2 \rightarrow er_1 \cap er_2 = er_1\ \&\ er_2 \end{array} \right.$$

En effet d'après la définition de l'intersection des expressions régulières `A & B` est l'intersection des langages `A` et `B`. C'est à dire l'ensemble des chaînes de caractères qui sont dans `A` *et* dans `B`.

L'opérateur $\cap$ sur les expressions régulière est associatif et commutatif. En effet soient $er_1, er_2, er_3 \in ExpressionRégulière$ nous avons alors les relations suivantes :

$$\begin{array}{c} er_1\ \&\ er_2 \equiv er_2\ \&\ er_1 \\ (er_1\ \&\ er_2)\ \&\ er_3 \equiv er_1\ \&\ (er_2\ \&\ er_3) \end{array} \tag{7.2}$$

7.2.1.4 Définition de l'opérateur $\cap$ entre deux collections de types

L'opération $\cap$ sur deux collections ordonnées de types de paramètres calcule la collection ordonnée commune de type si elle existe. L'intersection $\cap$ entre deux types est requise par le modèle et doit être stable et commutative.

Pour définir l'opération $\cap$ sur deux collections ordonnées de types de paramètres, il faut examiner plusieurs cas. Si les deux collections ont la même longueur alors on calcule pour chaque élément l'intersection des types. Si les deux collections n'ont pas la même taille mais qu'une des deux possède un métacaractère infini en dernière position alors on effectue sur la taille de la collection la plus petite une intersection des types. Sinon le résultat est une collection vide.

L'opération $\cap$ sur des collections ordonnées de types des paramètres d'espaces de fonctions se définit ainsi :

$$\cap : \left| \begin{array}{l} Collection < Type > , Collection < Type > \rightarrow Collection < Type > \\ \\ ct_1, ct_2 \rightarrow ct_1 \cap ct_2 = \quad ct \in Collection < Type > \text{ tq :} \\ \quad \text{Si } (ct_1.size == ct_2.size) \\ \quad \text{alors } \forall i \text{ tq } 0 < i < ct_1.size \text{ faire } ct.add(ct_1[i].type \cap ct_2[i].type) \\ \quad \text{sinon si } (ct_1.last == \text{'..'} \mid\mid ct_2.last == \text{'..'}) \\ \quad\quad \text{alors} \quad \forall i \text{ tq } 0 < i < min(ct_1.size, ct_2.size) \\ \quad\quad\quad\quad \text{faire } ct.addLast(ct_1[i].type \cap ct_2[i].type) \\ \quad\quad\quad \text{si} \quad ct_1.size == min(ct_1.size, ct_2.size) \\ \quad\quad\quad \text{alors} \quad \forall i \text{ tq } ct_1.size < i < ct_2.size \\ \quad\quad\quad\quad\quad \text{faire } ct.add(ct_2[i].type) \\ \quad\quad\quad \text{sinon} \quad \forall i \text{ tq } ct_2.size < i < ct_1.size \\ \quad\quad\quad\quad\quad \text{faire } ct.add(ct_1[i].type) \\ \quad \text{sinon } ct = \emptyset \end{array} \right.$$

La preuve de la commutativité de l'opérateur $\cap$ sur les collections ordonnées de paramètres se montre par sa définition et par la commutativité de l'opérateur $\cap$ sur les types. La preuve de la stabilité de l'opérateur $\cap$ se fait en deux cas distincts.

On examine d'abord le cas où les trois collections sont de même taille. Soient $ct_1, ct_2, ct_3 \in Collection < Type >$ et $ct_1.size = ct_2.size = ct_3.size$ alors par définition de $\cap$ sur les types : $(ct_1 \cap ct_2) \cap ct_3 \equiv ct_1 \cap (ct_2 \cap ct_3)$

Dans le cas où les collections sont de tailles différentes alors l'opérateur $\cap$ renvoie à chaque fois la collection la plus longue composée de l'intersection des types jusqu'à la fin de la collection la plus courte et complète par les types de la collection la plus longue. De même que précédemment, on a donc bien : $(ct_1 \cap ct_2) \cap ct_3 \equiv ct_1 \cap (ct_2 \cap ct_3)$.

Soient $ct_1, ct_2, ct_3 \in Collection < Type >$, on peut donc écrire de manière générale :

$$\begin{array}{c} ct_1 \cap ct_2 \equiv ct_2 \cap ct_1 \\ (ct_1 \cap ct_2) \cap ct_3 \equiv ct_1 \cap (ct_2 \cap ct_3) \end{array} \tag{7.3}$$

D'après les équation 6.1, 7.2 et 7.3 on peut déduire la commutativité et à la stabilité de l'opération $\cap$ sur les signatures de fonctions. Soient $s_1, s_2, s_3 \in SignatureDeFonctions$, on écrit donc les équations suivantes :

$$\begin{array}{c} s_1 \cap s_2 \equiv s_2 \cap s_1 \\ (s_1 \cap s_2) \cap s_3 \equiv s_1 \cap (s_2 \cap s_3) \end{array} \tag{7.4}$$

De plus de par la définition de l'opérateur $\cap$ sur les espaces de fonctions et les équations 7.1 et 7.4 on déduit la commutativité et la stabilité de l'opération $\cap$ sur les espaces de fonctions. Soient $e_1, e_2, e_3 \in EspaceDeFonctions$, on écrit donc :

$$\boxed{\begin{array}{c} e_1 \cap e_2 \equiv e_2 \cap e_1 \\ (e_1 \cap e_2) \cap e_3 \equiv e_1 \cap (e_2 \cap e_3) \end{array}} \tag{7.5}$$

7.2.2 Définition de l'opérateur $\setminus$ entre deux espaces de fonctions

L'opération $\setminus$ sur les espaces de fonctions se définit ainsi :

$$
\setminus : \left|
\begin{array}{l}
EspaceDeFonctions,\ EspaceDeFonctions \rightarrow\ EspaceDeFonctions \\
\\
e_1, e_2 \rightarrow e_1 \setminus e_2 =\ e \in EspaceDeFonctions \text{ tq :} \\
e.signatureDeFonctions = e_1.signatureDeFonctions \\
e.signatureDeFonctions.signaturesExclues.add(e_2.signatureDeFonctions) \\
e.métaobjets = e_1.métaobjets
\end{array}
\right.
$$

L'espace de fonction résultant est construit en retirant toutes les signatures de fonctions de e_2 à e_1 et en ne conservant que les métaobjets de e_1.

Soient $e_1, e_2, e_3\ \in EspaceDeFonctions$, alors :

$$\boxed{(e_1 \setminus e_2) \setminus e_3 \equiv (e_1 \setminus e_3) \setminus e_2} \tag{7.6}$$

En effet pour les attributs `signatureDeFonctions` et `métaobjets` ce sont bien ceux de e_1 qui sont conservés. Quand à la collection de signatures exclues elle est composée des signatures exclues de e_1, e_2 et e_3 comme c'est une collection non ordonnée cela démontre l'équation 7.6.

7.2.3 Définition de l'opérateur $\setminus$ entre un espace de fonctions et une collection d'espaces de fonctions

L'opération $\setminus$ entre un espace de fonctions et une collection d'espaces de fonctions se définit en ajoutant aux signatures exclues de l'espace de fonctions résultant toutes les signatures de fonctions de la collection d'espaces de fonctions.

$$
\setminus : \left|
\begin{array}{l}
EspaceDeFonctions,\ Collection < EspaceDeFonctions > \rightarrow\ EspaceDeFonctions \\
\\
e_1, ce_2 \rightarrow e_1 \setminus ce_2 =\ e \in EspaceDeFonctions \text{ tq :} \\
e.signatureDeFonctions = e_1.signatureDeFonctions \\
\forall i \text{ tq } 0 < i < ce_2.size \text{ alors} \\
\quad e.signatureDeFonctions.signaturesExclues.add(ce_2[i].signatureDeFonctions) \\
\quad e.métaobjets = e_1.métaobjets
\end{array}
\right.
$$

7.2.4 Unicité et stabilité de l'opérateur $\rho_{espacesDeFonctions}$

Après avoir détaillé les opérations sur les espaces de fonctions montrons que l'opération $\rho_{espacesDeFonctions}$ renvoie une solution unique. La démonstration se fait en prouvant l'associativité de l'opération ρ_{cedf}. Pour cela on démontre que quelques soient E une collection d'espaces de fonctions bien partitionnée et ef_1 et ef_2 deux espaces de fonctions alors $\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E)) \equiv \rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E))$.

Considérons d'abord $\rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E))$:

Soient $ef_1 \in EspaceDeFonctions$ et $E = \{e_1, ..., e_n\}$ tq $\forall i\ e_i \in EspaceDeFonctions$ et $\forall i \neq j\ e_i \cap e_j = \emptyset$ alors $\rho_{cedf}(ef_1, E) = \{\forall i \leq n \mid e_i \setminus ef_1, e_i \cap ef_1, ef_1 \setminus \bigcup E\}$

Soit $ef_2 \in EspaceDeFonctions$
$\rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E)) =$
$\{\forall i \leq n \mid (e_i \setminus ef_1) \setminus ef_2, (e_i \setminus ef_1) \cap ef_2,$
$(e_i \cap ef_1) \setminus ef_2, (e_i \cap ef_1) \cap ef_2,$
$(ef_1 \setminus \bigcup E) \setminus ef_2, (ef_1 \setminus \bigcup E) \cap ef_2,$
$ef_2 \setminus \{(e_i \setminus ef_1) \cup (e_i \cap ef_1) \cup (ef_1 \setminus \bigcup E)\}\}$

Considérons maintenant $\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E))$:

Soient $ef_2 \in EspaceDeFonctions$ et $E = \{e_1, ..., e_n\}$ tq $\forall i\ e_i \in EspaceDeFonctions$ et $\forall i \neq j\ e_i \cap e_j = \emptyset$ alors $\rho_{cedf}(ef_2, E) = \{\forall i \leq n \mid e_i \setminus ef_2, e_i \cap ef_2, ef_2 \setminus \bigcup E\}$

Soit $ef_1 \in EspaceDeFonctions$
$\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E)) =$
$\{\forall i \leq n \mid (e_i \setminus ef_2) \setminus ef_1, (e_i \setminus ef_2) \cap ef_1,$
$(e_i \cap ef_2) \setminus ef_1, (e_i \cap ef_2) \cap ef_1,$
$(ef_2 \setminus \bigcup E) \setminus ef_1, (ef_2 \setminus \bigcup E) \cap ef_1,$
$ef_1 \setminus \{(e_i \setminus ef_2) \cup (e_i \cap ef_2) \cup (ef_2 \setminus \bigcup E)\}\}$

Pour montrer l'équivalence $\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E)) \equiv \rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E))$, nous montrons l'égalité des ensembles résultats. Pour cette démonstration nous utilisons les équivalences suivantes issues des équations 7.5 et 7.6, soient $e_1, e_2, e_3 \in EspaceDeFonctions$ alors :

$$\begin{aligned}(e_1 \setminus e_2) \setminus e_3 &\equiv (e_1 \setminus e_3) \setminus e_2\\ (e_1 \setminus e_2) \cap e_3 &\equiv (e_1 \cap e_3) \setminus e_2\\ (e_1 \setminus e_2) \cap e_3 &\equiv (e_3 \setminus e_2) \cap e_1\end{aligned}$$

Ce qui nous permet de déduire d'après les ensembles produits par $\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E))$ et $\rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E))$ que :

$$\begin{aligned}(e_i \setminus ef_1) \setminus ef_2 &= (e_i \setminus ef_2) \setminus ef_1\\ (e_i \setminus ef_1) \cap ef_2 &= (e_i \cap ef_2) \setminus ef_1\\ (e_i \cap ef_1) \setminus ef_2 &= (e_i \setminus ef_2) \cap ef_1\\ (e_i \cap ef_1) \cap ef_2 &= (e_i \cap ef_2) \cap ef_1\\ (ef_1 \setminus \bigcup E) \cap ef_2 &= (ef_2 \setminus \bigcup E) \cap ef_1\\ ef_2 \setminus \{(e_i \setminus ef_1) \cup (e_i \cap ef_1) \cup (ef_1 \setminus \bigcup E)\} &= (ef_2 \setminus \bigcup E) \setminus ef_1\\ (ef_1 \setminus \bigcup E) \setminus ef_2 &= ef_1 \setminus \{(e_i \setminus ef_2) \cup (e_i \cap ef_2) \cup (ef_2 \setminus \bigcup E)\}\end{aligned}$$

Et que donc l'opération ρ_{cedf} produit un résultat unique :

$$\boxed{\rho_{cedf}(ef_1, \rho_{cedf}(ef_2, E)) \equiv \rho_{cedf}(ef_2, \rho_{cedf}(ef_1, E))} \quad (7.7)$$

Ce qui nous permet de par la définition de $\rho_{espacesDeFonctions}$ et l'équation 7.7 de déduire que l'opération $\rho_{espacesDeFonctions}$ respecte la propriété d'unicité du résultat. Soit d et d_1 deux DAC tel que $d_1 = \rho(d)$ alors par construction nous avons bien :

$$\forall\ e_i, e_j \in d_1.espacesDeFonctions\ ,\ \text{si}\ i \neq j\ \text{alors}\ e_i \cap e_j = \emptyset$$

La stabilité est issue de la définition de l'opération ρ_{cedf} et des définitions des opérations $\cap$ et $\setminus$ sur les espaces de fonctions. En effet par construction de la collection d'espaces de fonctions résultante l'opération ρ_{cedf} ne réduit pas les espaces de fonctions atteignables. En effet soient t un appel de fonctions de type `Appel` et e_1, e_2 deux espaces de fonctions, alors par définition des opérations :

$$\text{si}\ t \in \mathcal{L}(e_1)\ \text{ou}\ t \in \mathcal{L}(e_2)\ \text{alors}\ t \in \mathcal{L}(e_1 \setminus e_2)\ \text{ou}\ t \in \mathcal{L}(e_1 \cap e_2)\ \text{ou}\ t \in \mathcal{L}(e_2 \setminus e_1)$$

Ce qui nous permet de par la définition de $\rho_{espacesDeFonctions}$ et l'équation 7.7 de déduire que l'opération $\rho_{espacesDeFonctions}$ respecte la propriété de stabilité. Soit $d \in DAC$ tel que $d_1 = \rho(d)$ alors par construction nous avons bien :

$$\forall\ t \in Appel\ \text{tq}\ t \in \mathcal{L}(d.espacesDeFonctions)\ \text{alors}\ t \in \mathcal{L}(d_1.espacesDeFonctions)$$

7.3 Unicité et stabilité de l'opérateur ρ

La composition des évolutions structurelles (cf. 7.1) et comportementales (cf. 7.2) fournissent quand il n'y a pas de conflit un résultat unique et sont des opérations stables. Nous avons donc l'unicité et la stabilité des opérations $\rho_{attributs}$, $\rho_{méthodes}$ et $\rho_{espacesDeFonctions}$ quand elles ne rencontrent pas de conflits.

L'opérateur ρ applique successivement les trois opérations $\rho_{attributs}$, $\rho_{méthodes}$ et $\rho_{espacesDeFonctions}$ à une description abstraite de composant. Par définition de l'opérateur ρ et par les démonstrations d'unicité et de stabilité de ces trois opérations, l'opérateur ρ possède donc les propriétés d'unicité et de stabilité.

La définition des propriétés de ρ sont donc :

$$\rho : \left| \begin{array}{l} DAC \ \rightarrow\ DAC \cup Conflit \\ \\ d \rightarrow \rho(d) = \quad d_1 \in DAC\ \text{si}\ Conflit = \emptyset\ \text{tq} : \\ \\ \qquad \forall\ a_i, a_j \in d_1.attributs\ ,\ \text{si}\ i \neq j\ \text{alors}\ a_i \neq a_j \\ \qquad \forall\ i,\ d_1.attributs.contains(d.attributs[i]) \\ \\ \qquad \forall\ m_i, m_j \in d_1.méthodes\ ,\ \text{si}\ i \neq j\ \text{alors}\ m_i \neq m_j \\ \qquad \forall\ i,\ d_1.méthodes.contains(d.méthodes[i]) \\ \\ \qquad \forall\ e_i, e_j \in d_1.espacesDeFonctions\ ,\ \text{si}\ i \neq j\ \text{alors}\ e_i \cap e_j = \emptyset \\ \qquad \forall\ t \in Appel\ \text{tq}\ t \in \mathcal{L}(d.espacesDeFonctions) \\ \qquad \text{alors}\ t \in \mathcal{L}(d_1.espacesDeFonctions) \end{array} \right.$$

Conclusion

Dans ce chapitre nous avons explicité l'opération de composition ρ sur les descriptions abstraites de composants ainsi que ses propriétés d'unicité et de stabilité.

Nous avons formalisé l'ensemble des opérations de compositions sur les évolutions structurelles et comportementales et prouvé les propriétés d'unicité et de stabilité.

Notons que la composition traite tous les éléments d'une description abstraite de composant même ceux qui sont vides par définition, ce qui lève des cas d'erreurs sur des éléments vides et donc non valides.

Dans le chapitre suivant nous étudions la correspondance entre les descriptions abstraites de composants qui ont été composées par l'opérateur ρ et les informations des composants concrets.

CHAPITRE 8

Le modèle des composants concrets et les phases de correspondance

Dans ce chapitre nous étudions le modèle de composants concrets. Ce modèle se positionne entre le modèle des descriptions abstraites de composants et les composants de l'application. D'une part les informations nécessaires sont extraites des composants de l'application et ajoutées aux composants concrets, et d'autre part les descriptions abstraites de composants sont projetées dans les composants concrets. Cette étape de correspondance permet :

- de vérifier qu'il n'y pas de conflits entre les informations issues du composants de l'application et les évolutions structurelles (conflits de surcharge et « respect des types »).
- de déplacer les métaobjets des espaces de fonctions vers les méthodes concrètes.
- de déterminer les objets d'implantation dans lesquels les évolutions structurelles et comportementales vont être projetées.
- de transformer les règles de réécriture en appels effectifs.

Dans un premier temps nous présentons le modèle de composants concrets. Puis nous détaillons l'ensemble des étapes (cf. figure 8.1) : l'extraction des informations des composants de l'application, la transformation des évolutions structurelles puis comportementales des descriptions abstraites de composants, la détermination des objets d'implantation et la transformation des règles de réécriture en appels effectifs. Enfin nous discutons la projection des composants concrets en composants de l'application qui ont été préparés pour la plateforme cible.

8.1 Le modèle des composants concrets

Le type `ComposantConcret` décrit un composant à l'aide d'un nom, d'une collection d'attributs concrets et d'une collection de méthodes concrètes. Les types `Attribut-Concret` et `MéthodeConcrète` sont dérivés des types `Attribut` et `Méthode` du modèle des descriptions abstraites de composants.

Dans ce modèle ce sont les méthodes concrètes qui vont jouer le rôle principal. Elles vont dans un premier temps porter les métaobjets qui représentent les évolutions comportementales. Dans un second temps, elles vont porter les objets d'implan-

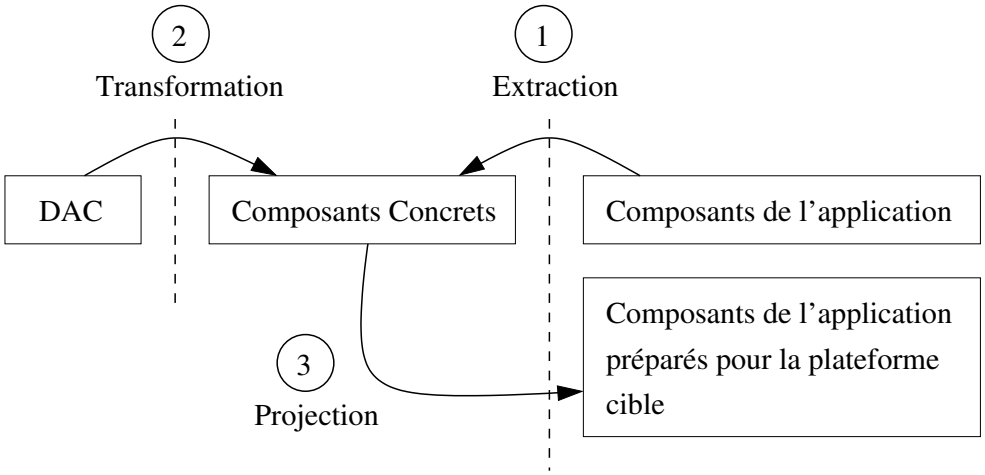

FIG. 8.1 – Phases de correspondances

tation qui sont susceptibles d'être impactés par les évolutions comportementales. Les objets d'implantation représentent les différents objets qui composent un composant dans la plateforme à cible. Il faut donc disposer d'une représentation de la plateforme cible qui identifient les objets d'implantation - notons que ces représentations sont souvent inexistantes même si l'approche MDA va favoriser leur développement. Les objets d'implantation sont dépendants de la plateforme dans laquelle les évolutions vont être projetées. En fonction des plateformes cibles, il pourra s'agir de *Proxys*, *Intercepteurs*, *Implantations*, *Interfaces*, etc.

Une table de correspondances permet de lier d'une part les objets d'implantation et les visibilités et d'autre part les objets d'implantation et les métaobjets. Nous définissons le type `RelationOIV` qui représente les relations entre les objets d'implantation et les visibilités. Ces relations sont utilisées lors de la phase d'extraction d'informations des composants de l'application. Et nous définissons le type `RelationOIM` qui représente les relations entre les objets d'implantation et les métaobjets qui sont utilisés lors de la phase de transformations des descriptions abstraites de composants. Un objet d'implantation joue un rôle de contrôle au niveau de la plateforme. En conséquence un métaobjet se projettera dans un objet d'implantation. Plusieurs métaobjets peuvent se projeter dans un même objet d'implantation. Par contre un métaobjet ne peut pas se projeter dans plusieurs objets d'implantation, dans ce cas la modélisation du composant utilisée est considérée comme insuffisante et doit être enrichie pour discrétiser le contrôle

Les relations sont dépendantes des objets d'implantation et donc de la plateforme à composants cible. La table de correspondances possède une interface d'interrogation pour sélectionner à partir d'un élément tous les autre éléments en relation.

La figure 8.2 montre le modèle des composants concrets.

Nous allons maintenant discuter de l'extraction des informations des composants de l'application qui est la première phase de l'étape de correspondance.

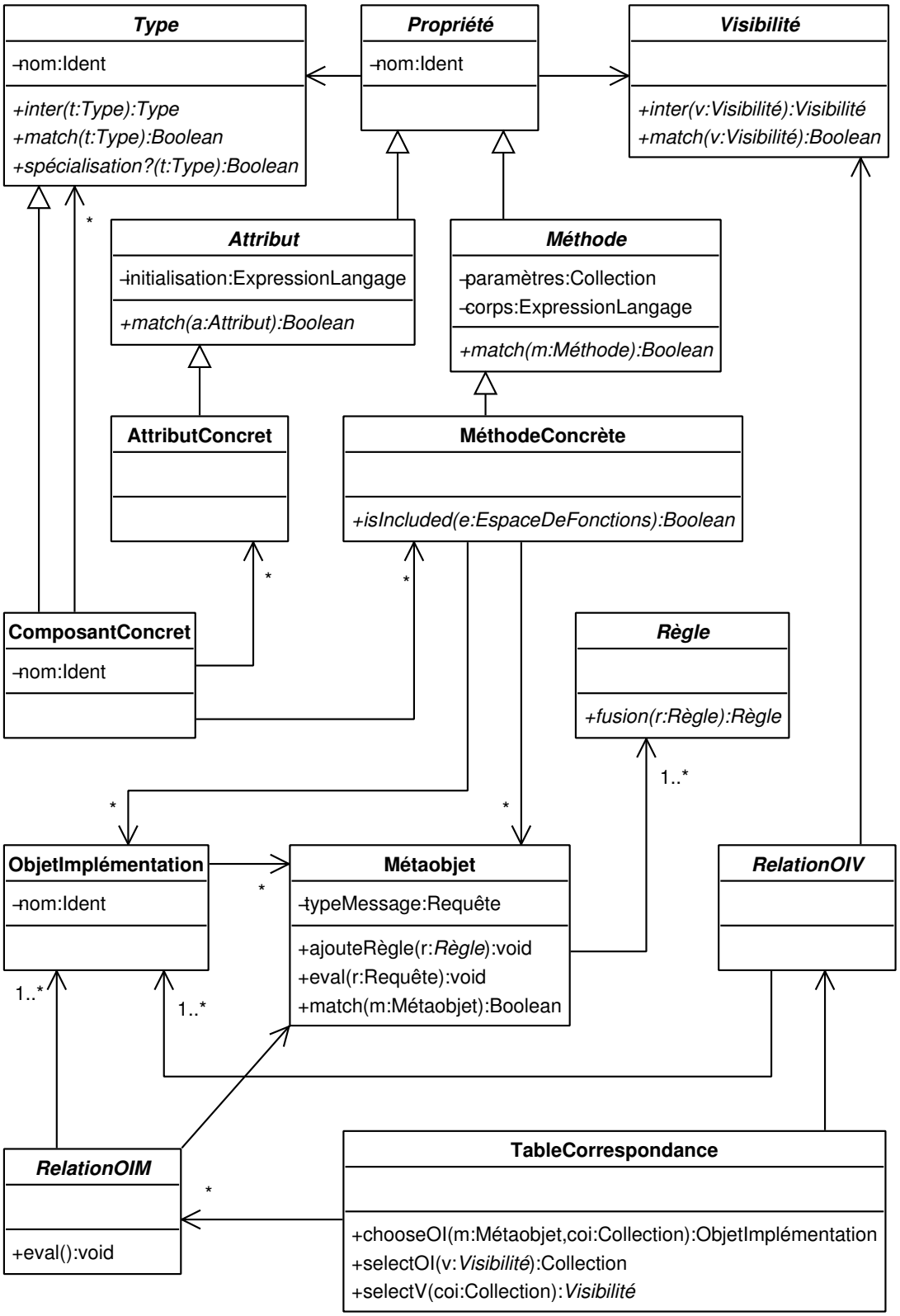

FIG. 8.2 – Le modèle de composants concrets

8.2 Extraction des informations des composants de l'application

La première phase de l'étape de correspondance extrait les informations des composants de l'application vers les composants concrets. Les composants de l'application à analyser sont ceux désignés dans le fichier de configuration. L'analyse dépend de la plateforme à composants cible et des informations que l'on veut analyser.

Exemple :
Ainsi l'analyse d'un composant pour la plateforme à composants JOnAS peut porter sur l'EJB Object, l'objet d'interposition, le stub et le squelette par exemple. Une autre analyse peut porter uniquement sur l'interface Remote et l'EJB Object.

L'analyse des composants de l'application extrait des informations sur les attributs et sur les méthodes. Suivant où se trouve la propriété et ses informations de visibilité dans la plateforme, l'analyse va en déduire une visibilité issue du système externe de visibilité à l'aide des relations `RelationOIV` de la table de correspondances.

Exemple :
Dans le premier cas d'analyse précédent, une propriété définie et implémentée uniquement au niveau de l'objet d'interposition mais non décrite par l'interface Remote aura dans notre système de visibilité (cf. figure 9) au niveau du composant concret la visibilité : *service metier*.
Dans le deuxième cas d'analyse, une propriété définie au niveau de l'EJB Object mais non référencée par l'interface aura dans notre système de visibilité au niveau du composant concret une visibilité : *private metier composant*.

Cette opération est basée sur la relation `RelationOIV` de la table de correspondances entre la localisation d'une propriété dans le composant de l'application et une visibilité du système de visibilité. La notion de localisation est complexe. En effet une méthode peut être présente dans plusieurs des objets qui forment le composant, avec un nommage différent. L'analyse doit exprimer la localisation des propriétés sous formes d'objets d'implantation pour ensuite pouvoir interroger la table de correspondance sur les relations `RelationOIV` et déduire la visibilité.

Nous ne proposons pas de mise en œuvre de l'analyse car la dépendance aux plateformes est trop forte. Mais des travaux autour de la métamodélisation indépendante des langages [106] permettent de s'abstraire de l'analyse directe du code et d'utiliser un modèle d'interrogation pour faire l'analyse.

Une fois la phase d'extraction des informations des composants de l'application terminée, les composants concrets sont une représentation des composants de l'application. La phase suivante permet de transformer les descriptions abstraites de composants vers les composants concrets.

8.3 Transformation des descriptions abstraites de composants vers les composants concrets

Tout d'abord le type du composant concret est validé par rapport aux types que porte la description abstraite de composant. Cette validation se fait en vérifiant que

le type du composant concret est une spécialisation de chacun des types de la description abstraite de composant. L'opération pour tester si un type est une spécialisation d'un autre est issue du système de typage. Elle se nomme `spécialisation ?`.

$$
projetteTypes : \left| \begin{array}{l} DAC,\ ComposantConcret \rightarrow\ ComposantConcret \cup Conflit \\ \\ d, c \rightarrow\ projetteTypes(d, c) \equiv \\ \quad \forall\ t \in d.types \text{ faire :} \\ \quad \text{Si } not(c.type.spécialisation?(t)) \\ \quad \text{Alors } new\ Conflit(t); \end{array} \right.
$$

Ensuite, les descriptions abstraites de composants sont projetées dans les composants concrets, d'abord les évolutions structurelles puis les évolutions comportementales.

Cette étape permet de détecter des conflits entre les propriétés des descriptions abstraites de composants et les propriétés des composants de l'application. Donc avant d'ajouter une propriété dans le composant concret à partir de la description abstraite de composant nous vérifions par la méthode `match` définie sur les attributs et les méthodes qu'il n'y a pas de conflits avec les propriétés du composant concret extraites des composants de l'application.

On écrit alors l'opération `projetteAttributs` qui permet de projeter tous les attributs d'une description abstraite de composant dans un composant concret :

$$
projetteAttributs : \left| \begin{array}{l} DAC,\ ComposantConcret \rightarrow\ ComposantConcret \cup Conflit \\ \\ d, c \rightarrow\ projetteAttributs(d, c) \equiv \\ \quad \forall\ a \in d.attributs \text{ faire :} \\ \quad \text{Si } c.attributs.contains(a) \\ \quad \text{Alors } new\ Conflit(a); \\ \quad \text{Sinon } c.addAttribut(a); \end{array} \right.
$$

De même l'opération `projetteMéthodes` qui permet de projeter toutes les méthodes d'une description abstraite de composant dans un composant concret :

$$
projetteMéthodes : \left| \begin{array}{l} DAC,\ ComposantConcret \rightarrow\ ComposantConcret \cup Conflit \\ \\ d, c \rightarrow\ projetteMéthodes(d, c) \equiv \\ \quad \forall\ m \in d.méthodes \text{ faire :} \\ \quad \text{Si } c.méthodes.contains(m) \\ \quad \text{Alors } new\ Conflit(m); \\ \quad \text{Sinon } c.addMéthode(m); \end{array} \right.
$$

Une fois les évolutions structurelles projetées dans le composant concret, il faut projeter les évolutions comportementales. La projection des évolutions comportementale se fait en identifiant pour chaque méthode concrète si elle est affectée par une évolution comportementale. Pour chaque méthode concrète, nous testons par la méthode `isIncluded` si elle est dans le langage défini par un espace de fonctions. Quand un espace de fonctions correspond alors les métaobjets de l'espace de fonctions sont

dupliqués sur la méthode concrète. Du fait que tous les espaces de fonctions sont tous disjoints (cf. chapitre 7), il n'y a qu'un seul espace de fonctions qui peut correspondre. Cette opération ne peut pas générer de conflit. Par contre un espace de fonctions peut ne pas être projeté.

$$
projetteEspacesDeFonctions : \left|
\begin{array}{ll}
 & DAC,\ ComposantConcret \rightarrow\ ComposantConcret \\
 & \\
d, c \rightarrow & projetteEspacesDeFonctions(d, c) \equiv \\
 & \forall\ m \in c.méthodes \\
 & \& \ \forall e \in d.espacesDeFonctions \text{ faire :} \\
 & \text{Si } m.isIncluded(e); \\
 & \text{Alors } m.métaobjets.concat(e.métaobjets);
\end{array}
\right.
$$

A la fin de cette phase les composants concrets réunissent les informations issues des composants de l'application ainsi que les évolutions issues des descriptions abstraites de composants. La phase suivante permet de raffiner les composants concrets et de sélectionner les objets d'implantation qui vont être impactés par les différentes évolutions.

8.4 Raffinement des composants concrets

Cette étape de raffinement permet de répartir les évolutions sur les objets d'implantation. Elle se divise en trois phases. La première phase consiste à sélectionner les objets d'implantation qui vont être impactés, puis à passer les métaobjets des méthodes concrètes vers les objets d'implantation et enfin à transformer les règles de réécriture en appels effectifs.

8.4.1 Sélection des objets d'implantation

La sélection des objets d'implantation renvoie une collection qui contient les objets d'implantation sur lesquels vont être appliqués les évolutions. Cette sélection se base sur les informations de visibilités des méthodes. Nous utilisons la relation `RelationOIV` de la table de correspondances :

$$
selectOI : \left|
\begin{array}{ll}
 & ComposantConcret \rightarrow\ ComposantConcret \\
 & \\
c \rightarrow & selectOI(c) \equiv \\
 & \forall\ m \in c.méthodes \text{ faire :} \\
 & Collection < ObjetImplémentation > coi = \\
 & \qquad TableCorrespondance.selectOI(m.visibilité); \\
 & m.objetsImplémentation.concat(coi);
\end{array}
\right.
$$

Pour chaque méthode concrète et suivant sa visibilité, une collection d'objets d'implantation est créée et associée à la méthode concrète. Comme nous l'avons déjà discuté la relation `RelationOIV` de la table de correspondances est spécifique à la plateforme. Nous donnons des exemples de relations `RelationOIV` dans la partie III.

8.4.2 Annotations des objets d'implantation par les métaobjets

Cette étape permet de répartir les métaobjets sur les objets d'implantation qui ont été sélectionnés à l'étape précédente. Ainsi les contrôles que définissent les métaobjets sont déplacés sur les objets d'implantation qu'ils vont contrôler. Nous utilisons la relation `RelationOIM` de la table de correspondances :

$$
\begin{array}{l|l}
 & ComposantConcret \rightarrow\ void \\
 & \\
annotateOI : & \begin{array}{ll} c \rightarrow & annotateOI(c) \equiv \\ & \forall\ m \in c.méthodes \\ & \&\ \forall\ o \in m.métaobjets \text{ faire :} \\ & \quad ObjetImplémentation\ oi = \\ & \qquad TableCorrespondance.chooseOI(o, m.objetsImplémentation) \\ & \quad oi.setMétaobjet(o) \end{array}
\end{array}
$$

Chaque métaobjet de chaque méthode concrète choisit à partir de la table de correspondances l'objet d'implantation de la méthode concrète sur lequel il va se placer. Plusieurs métaobjets peuvent se retrouver sur le même objet d'implantation. Comme les métaobjets sont ordonnés séquentiellement par le plan de base, il n'y a pas de problème de combinaison.

8.4.3 Transformation des règles de réécritures en appels effectifs

Dans cette phase les règles de réécritures sont transformées en appels effectifs. On utilise la fonction d'évaluation de la relation `RelationOIV`. Cela permet de résoudre les liens vers les objets concrets du système. Cette opération est dépendante de la plateforme cible et est implémentée par la fonction `eval` de la relation `RelationOIV`.

8.5 Génération de code

Dans cette étape, à partir des appels effectifs, le code est généré dans les composants de l'application. Ce processus est l'inverse du processus d'analyse de la phase d'extraction des informations des composants de l'application.

Cette étape de génération peut être envisagée de deux manières différentes. Dans le premier cas les objets des composants de l'application ont déjà été préparés par les outils de la plateforme à composants cible. Dans ce cas il faut faire attention de ne pas décabler les différents appels de méthodes et les liaisons entre les objets du composants de l'application. Certains contrôles des métaobjets sont alors présents et il ne faut pas les remplacer. Dans un deuxième cas les outils de la plateforme à composants n'ont pas été utilisés et cette phase de génération de code doit générer toutes les classes et tout le code du composant. Dans ce deuxième cas notre approche remplace le générateur habituel de la plateforme à composants.

Nous ne détaillons pas plus avant la génération de code car elle est dépendante de la plateforme cible et est gérée de manière ad hoc.

Conclusion

Dans ce chapitre nous avons abordé la phase de correspondance et le modèle des composants concrets. Ce modèle comme le modèle des descriptions abstraites de composants repose sur des systèmes externes. Les objets d'interpositions sont une modélisation des composants de l'application et dépendent de la plateforme à composant cible. Les diverses opérations sont laissées à la charge des systèmes externes et de relation de la table de correspondances des objets d'implantation. Cette table de correspondance qui permet de faire remonter l'information des composants de l'application puis de générer le code est dépendante de la plateforme à composants cible et de sa modélisation par les objets d'implantation. Nous montrons un exemple de table de correspondance pour une modélisation de la plateforme JOnAS dans la partie III.

Récapitulatif

Dans cette partie nous avons décrit et formalisé notre approche de l'intégration de services. Nous avons tout d'abord présenté notre scénario de l'intégration de services dans sa globalité au travers de deux exemples simples.

Dans un deuxième temps nous avons décrit et formalisé notre modèle des descriptions abstraites de composants qui permet de décrire les évolutions à appliquer à un composant de manière abstraite et indépendamment des plateformes à composants. Nous avons pour cela décrit l'utilisation de systèmes externes qui permettent cette abstraction. Nous avons ensuite décrit le modèle des intégrateurs qui permet de paramétrer les évolutions et de manipuler le modèle des descriptions abstraites de composants.

Ensuite nous avons détaillé et formalisé la composition des évolutions. Nous avons montré que l'opération de composition est stable et fournit un résultat unique quand il n'y a pas de conflit.

Enfin nous avons décrit l'étape de correspondance qui se base sur le modèle des composants concrets. Nous avons montré comment les informations issues des composants de l'application sont extraites vers les composants concrets et comment les descriptions abstraites de composants sont transformées vers les composants concrets. Nous avons détaillé le raffinement des composants concrets qui permet de sélectionner les objets d'implantation dans lesquels les évolutions vont être projetées et de transformer les règles de réécriture pour résoudre les appels effectifs.

Dans la partie suivante nous mettons en œuvre notre modèle sur des exemples issus des services décrits dans la bibliographie. Nous commençons par compléter notre modèle par la description des systèmes externes que nous utilisons. Puis nous décrivons les exemples et l'ensemble du processus jusqu'à la génération de code pour la plateforme à composants JOnAS.

Troisième partie

APPLICATION À LA PLATEFORME JONAS

Dans cette partie...

Dans cette partie nous mettons en application notre modèle d'intégration de services à l'aide d'exemples. Nous décrivons d'abord les systèmes externes qui viennent compléter le modèle des descriptions abstraites de composants et celui des composants concrets. Ces systèmes externes sont adaptés à l'intégration de services dans les plateformes à composants que nous avons décrits dans la première partie. Pour chacun de ces systèmes externes nous détaillons les opérations qu'il doit implémenter pour se conformer à notre modèle défini précédemment. Ainsi nous détaillons les systèmes de types, de propriétés, de visibilités et de règles.

Puis nous décrivons et discutons quatre exemples d'intégration de services avec notre modèle. Nous montrons l'écriture de l'intégration d'un service de cache de méthodes qui permet de ne pas effectuer le calcul d'un résultat si un appel équivalent a déjà été fait. Nous montrons aussi l'écriture de l'intégrateur d'un service d'authentification qui modifie la signature des méthodes pour passer en plus un paramètre d'authentification, ainsi que l'intégrateur d'un service de chiffrement et aussi l'intégrateur d'un service de persistance. Nous montrons la décoration des descriptions abstraites de composants, ainsi que la composition, et la correspondance avec les composants concrets et finalement la projection. Nous montrons des cas de conflits et détaillons leur résolution et nous discutons aussi de l'implantation du modèle.

Enfin nous concluons cette thèse en récapitulant les points importants de notre approche ainsi que les limites de notre modèle. Nous donnons aussi les perspectives à ce travail.

CHAPITRE 9

Systèmes externes

Dans ce chapitre nous détaillons les systèmes externes qui viennent compléter notre modèle d'intégration de services pour réaliser l'intégration de services dans la plateforme JOnAS. Pour chacun de ces systèmes externes nous détaillons les opérations requises par notre modèle. Nous justifions nos choix pour ces systèmes externes vis-à-vis de la cible de projection. Nous discutons comment ces systèmes externes permettent de projeter les intégrations de services dans les différentes implantations de plateformes à composants que nous avons détaillées dans la première partie.

Nous prenons comme base le langage Java car il est le langage cible des projections. Nous utilisons le système de types de Java ainsi que le système de propriétés. Pour le système de visibilité nous détaillons un système qui correspond à une architecture générale de plateforme à composants. Pour le système de règles nous utilisons le langage ISL que nous étendons avec un métaopérateur d'interruption.

9.1 Système de types

L'implantation JOnAS ainsi que les autres implantations qui respectent la spécification EJB utilise Java comme langage d'implantation et de déploiement. Les composants ainsi que les objets qui les composent utilisent le système de typage de Java. Les compilateurs IDL Java permettent aussi d'utiliser le système de typage de Java avec les composants CORBA.

Nous utilisons le système de type à héritage simple de Java comme système externe de types que nous enrichissons d'un type vide. La figure 9.1 représente notre système de types.

Nous définissons l'opération d'intersection entre deux types. Cette opération renvoie le type commun s'il existe ou le type vide sinon. Pour répondre à la spécification de notre modèle, rappelons que l'opération d'intersection doit avoir les propriétés suivantes :

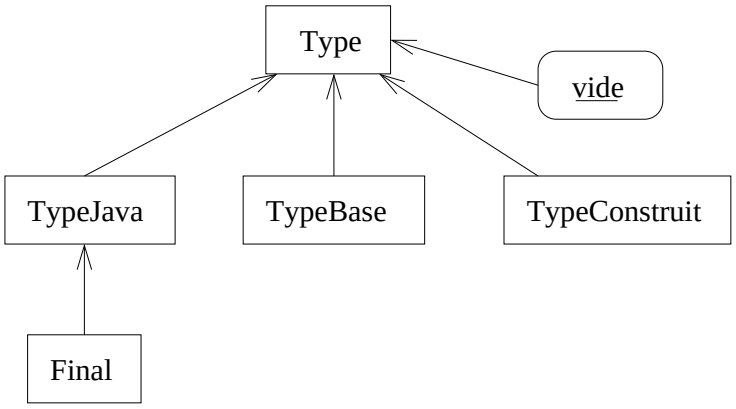

FIG. 9.1 – Système externe de types

$\forall t_1, t_2 \in Type$ alors $t_1 \cap t_2 \in Type$.
$\exists \perp \in Type$ tel que $t_1 \cap \perp \equiv \perp$.
L'opération $\cap$ est commutative : $t_1 \cap t_2 \equiv t_2 \cap t_1$.
L'opération $\cap$ est transitive : $(t_1 \cap t_2) \cap t_3 \equiv t_1 \cap (t_2 \cap t_3)$.

On explicite la fonction d'intersection en Prolog (cf. figure 9.2) tel que nous l'avons définie dans notre expérimentation. Nous nous munissons des opérations `typeBase`, `typeJava`, `final` qui permettent de connaître la nature d'un type dans le système externe de type. On définit le prédicat `spécialisable` qui répond vrai si un type n'est ni un type final, ni un type de base. Le prédicat `spécialisation` répond vrai si son premier paramètre est une spécialisation de son deuxième paramètre.

De manière générale l'intersection de deux types spécialisables correspond à un type qui vérifie les interfaces des deux types de l'intersection (que l'on note par `t1 & t2`). Pour calculer la spécialisation nous devons disposer d'une base de faits qui dépend du système de types du langage cible.

Nous rappelons que la définition de l'intersection et de l'équivalence nous sert au moment de la composition des espaces de fonctions.

L'équivalence `match` (notée $\equiv$) de deux type est vraie si l'intersection n'est pas vide. La figure 9.3 présente le code Prolog de l'équivalence.

$$t_1 \equiv t_2 \Leftrightarrow t_1 \cap t_2 \neq \perp$$

```
type_(T) :- typeBase(T).
type_(T) :- typeJava(T).
type_(T) :- typeConstruit(T).

type_(vide).

typeConstruit(and(T1,T2)) :- type_(T1), type_(T2).

%%Specialisable
specialisable(T1) :- type_(T1), not(typeBase(T1)), not(final(T1)).

specialisationConnue(T1,X) :- extends(X,T1).
specialisationConnue(T1,Y) :- implements(Y,T1).

%% Permet de tester si les types sont des spécialisations
specialisation(T1,T2) :- extends(T1,T2).
specialisation(T1,T2) :- implements(T2,T1).

specialisation(and(T1,_), T1).
specialisation(and(_,T2),T2).
specialisation(X,and(Z,Y)) :-
    specialisable(Z),Z\=X,
    specialisable(Y),
    specialisation(X,Z),
    specialisation(X,Y).
specialisation(T1,T2) :-
    specialisationConnue(T2,T3),specialisation(T1,T3).

intersection(T,T,T).
intersection(T1,T2,T1) :-
    specialisation(T1,T2).
intersection(T1,T2,T2) :-
    specialisation(T2,T1).
intersection(T1,T2,T3) :-
    specialisation(T3,T1),
    T3 \= and(_,T1),
    T3 \= and(T1,_),
    specialisation(T3,T2),
    not(plusgrand(T1,T2,T3)).

intersection(T1,T2,and(T1,T2)) :-
    specialisable(T1),
    specialisable(T2).
intersection(T1,T2,vide).

plusgrand(T1,T2,T3) :-
    specialisation(T3,T4),
    specialisation(T3,T4),
    specialisation(T4,T1),
    specialisation(T4,T2).
```

FIG. 9.2 – Calcul de la fonction d'intersection de types

```
equivalent(T,T).
equivalent(T1,T2) :-
    intersection(T1,T2,T3),
    vide \= T3.
```

FIG. 9.3 – Équivalence entre deux types

9.2 Système de propriétés

Pour répondre à la spécification de notre modèle le système externe de propriétés définit une opération d'équivalence sur les attributs et une opération d'équivalence sur les méthodes qui permettent de détecter des conflits dus à une surcharge non supportée par la plateforme à composant cible. Cette notion d'équivalence est dépendante du langage d'implantation et de la plateforme à composants. Pour la spécification EJB les règles de surcharge et de redéfinitions sont celles du langage Java. La spécification CCM impose des règles différentes puisque aucune surcharge ou redéfinition n'est acceptée.

Dans le cadre d'une projection dans JOnAS nous définissons l'équivalence des attributs et des propriétés suivant les règles de la spécification Java. Nous rappelons que les opérations d'équivalence sur les attributs et sur les méthodes permettent de détecter des conflits lors de la phase de composition des évolutions structurelles.

Ainsi deux attributs sont équivalents s'ils ont le même nom. L'opération `match` s'écrit alors :

$$match : \left| \begin{array}{l} \qquad Attribut,\ Attribut \rightarrow\ Boolean \\ \\ a_1, a_2 \rightarrow match(a_1, a_2) \equiv\ a_1.nom == a_2.nom \end{array} \right.$$

Deux méthodes sont équivalentes si leurs signatures ont le même nom, le même nombre de paramètres, et que chaque paramètre deux à deux a le même type. L'opération `match` entre deux méthodes s'écrit comme suit :

$$match : \left| \begin{array}{ll} & \qquad\qquad M\acute{e}thode,\ M\acute{e}thode \rightarrow\ Boolean \\ \\ m_1, m_2 \rightarrow & match(m_1, m_2) \equiv \\ & m_1.nom == m_2.nom \\ & \& \ m_1.param\grave{e}tres.size == m_2.param\grave{e}tres.size \\ & \& \ \forall\ i \in 0 < i < m_1.param\grave{e}tres.size \text{ alors} \\ & \qquad m_1.param\grave{e}tres[i].type == m_2.param\grave{e}tres[i].type \end{array} \right.$$

Notons que nous utilisons l'égalité stricte sur les types des paramètres ce qui correspond à la définition de la surcharge.

9.3 Système de visibilités

Dans l'étude que nous avons faite des plateformes à composants (cf. chapitre 3), nous avons vu que de manière générale et quelle que soit la plateforme à composants, plusieurs rôles sont définis au sein des plateformes à composants. Le but de la

visibilité est de fournir des indications quand au placement d'une propriété ou d'une évolution par rapport à un composant. Pour adhérer aux rôles définis par les plateformes à composants nous définissons une visibilité composite qui se compose de trois parties.

La première partie représente l'*accès extérieur*. Elle peut prendre trois valeurs : *public*, *service* ou *privé*. L'*accès extérieur* décrit dans le cas d'une propriété ajoutée, par qui la propriété peut être accédée. Dans le cas *public* la propriété est accessible depuis l'extérieur du composant. Dans le cas *service* la propriété est accessible depuis les composants de services. Dans le cas *privé* la propriété n'est utilisable que par le composant lui-même.

La seconde partie représente le *rôle*. Elle peut prendre deux valeurs : *home* ou *métier*. Le *rôle* décrit si la propriété ajoutée est destinée à être utilisée par la partie métier du composant ou bien par la fabrique de composants.

La troisième partie représente l'*accès aux données*. Elle peut prendre deux valeurs : *composant* ou *unknown*. L'*accès aux données* définit dans le cas d'une méthode ajouté qu'elle a besoin d'accéder aux données du composant, et dans le cas d'un attribut ajouté qu'il a besoin d'être considéré comme les autres attributs du composant.

Nous détaillons les combinaisons du système externe de visibilité et discutons brièvement de la projection dans la plateforme JOnAS à titre d'exemple. Certaines compositions ne sont pas possibles dans JOnAS mais sont valides dans d'autres plateformes. Quand la plateforme cible ne supporte pas une visibilité un conflit est détecté lors de la projection.

Voici l'ensemble des combinaisons du système de visibilités. Nous donnons aussi la projection JOnAS. Cette information de projection est stockée dans la table de correspondance.

- *public metier composant* : désigne une propriété accessible depuis l'extérieur du composant, utilisée par la partie métier du composant et qui a besoin d'accéder aux propriétés du composant.
  Dans ce cas la propriété est placée dans le bean, sa signature est ajoutée dans l'interface et l'objet d'interposition est modifié pour qu'il puisse aussi accepter cette propriété et faire transiter l'appel au bean.

- *public metier unknown* : désigne une propriété accessible depuis l'extérieur du composant, utilisée par la partie métier du composant et qui n'accède pas aux propriétés du composant.
  Dans ce cas la propriété est placée dans l'objet d'interposition, et sa signature est ajoutée dans l'interface.

- *public home composant* : désigne une propriété accessible depuis l'extérieur du composant, utilisée par la fabrique de composants et qui a besoin d'accéder aux données du composants.
  Dans JOnAS les messages ne transitent pas entre la home et le bean. Cette visibilité n'est donc pas projetable dans JOnAS.

- *public home unknown* : désigne une propriété accessible depuis l'extérieur, utilisée par la fabrique de composants et qui n'accède pas aux propriétés du composant.
  Dans ce cas la propriété est placée dans la home et l'interface de la home est

modifiée.

- *service metier composant* : désigne une propriété accessible depuis les composants de services (qui sont localisés dans le même espace que les composants métier), utilisée par la partie métier du composant et qui a besoin d'accéder aux propriétés du composant.
  Dans ce cas la propriété est placée dans le bean et l'objet d'interposition est modifié pour qu'il puisse aussi accepter cette propriété et faire transiter l'appel au bean.

- *service metier unknown* : désigne une propriété accessible depuis les composants de services, utilisée par la partie métier du composant et qui n'accède pas aux propriétés du composant.
  Dans ce cas la propriété est placée dans l'objet d'interposition.

- *service home composant* : désigne une propriété accessible depuis les composants de services, utilisée par la fabrique de composants et qui a besoin d'accéder aux données du composants.
  Cette visibilité n'est pas projetable dans JOnAS pour les mêmes raisons que précédemment.

- *service home unknown* : désigne une propriété accessible depuis les composants de services, utilisée par la fabrique de composants et qui n'accède pas aux données du composants.
  Dans ce cas la propriété est placée dans la home.

- *privé metier* : désigne une propriété accessible uniquement depuis le composant et utilisée par la partie métier du composant.
  Dans ce cas la propriété est placée dans le bean.

- *privé home* : désigne une propriété accessible uniquement par la fabrique de composants.
  Dans ce cas la propriété est placée dans la home.

9.4 Système de règles

Pour répondre aux contraintes imposées par notre modèle, le système externe de règles doit posséder une opération de composition commutative et associative, et doit aussi offrir une interface de réécriture des comportements qui permette des manipulations étendues.

Nous avons décidé d'utiliser pour cela le système de réécriture ISL qui possède ces deux propriétés. Le système de réécriture est bien adapté car son terme principale de réécriture est l'envoie de message. Comparé à d'autres systèmes [109], ISL offre l'ensemble des propriétés adéquates pour notre modèle : l'objet de réécriture est l'envoie de message, l'opération de composition est associative et commutative, les patterns de réécriture sont simple à mettre en place.

Nous utilisons donc le système de règles de réécriture ISL qui définit une opération

de composition commutative et associative, et détecte des conflits quand des règles ne sont pas composables. ISL permet de modifier le comportement associé à l'envoi d'un message. Une règle ISL est composée d'une partie gauche et d'une partie droite. La partie gauche correspond au message sur lequel la règle s'applique et la partie droite correspond au nouveau comportement associé au message. Le langage pour décrire le nouveau comportement est composé d'opérateurs que nous allons décrire de manière informelle. Le langage complet ainsi que la formalisation de la composition des règles ISL sont décrits dans [4]. Les opérateurs du langage ISL sont les suivants :

- *un appel du subséquent [_call()]* : permet d'exécuter le message sur lequel s'applique la règle.
- *un appel de méthode* : permet d'effectuer un appel de message sur un objet et de récupérer un résultat.
- *un envoi d'exception [throw]* : permet de lever une exception qui termine l'exécution de l'opération en cours d'exécution et remonte les opérations englobantes jusqu'à être capturer par un opérateur de capture d'exception soit terminer l'exécution du message.
- *une capture d'exception [try-catch]* : permet de capturer une exception levée dans une sous-opération et d'exécuter une opération alternative.
- *une attente [wait]* : permet d'étiqueter une opération et de mettre d'autres opérations en attente de la fin d'exécution de l'opération étiquetée.
- *une précédence [delegate]* : permet lors d'une composition de forcer la préemption d'une opération par rapport aux autres.
- *une séquence [;]* : permet d'exécuter une opération puis d'en exécuter une deuxième à la terminaison de la première.
- *un parallèle [| |]* : permet d'exécuter deux opérations de manière non ordonnée.
- *une conditionnelle [if-then-else]* : permet d'exécuter un test et suivant la valeur du résultat du test, d'exécuter soit l'opération associée à un résultat positif du test de la conditionnelle soit l'opération associée à un résultat négatif du test de la conditionnelle.

Exemple :

La règle 9.4 redéfinit le comportement d'un message `setColor` qui change la couleur d'un feu tricolore `t`. La règle ISL déclare l'envoi du message subséquent (le `_call`) et ensuite (la séquence ;) une conditionnelle (le `if-then-else`) qui porte sur la couleur du feu tricolore pour modifier la couleur du feu piéton `w`.

```
setColor(int color) :- _call(color) ;
                          if t.isRed() then w.setWalk()
                                       else w.setDontWalk()
```

FIG. 9.4 – Exemple de règle ISL avec une séquence et une conditionnelle

Exemple :

La règle 9.5 redéfinit le comportement d'un message `request` qui est une demande d'information à un serveur `master`. La règle ISL déclare l'envoi du message subséquent (le `_call`) et si une exception est levée (le `try-catch`) à cause du réseau alors on effectue la demande d'information sur un autre serveur `slave`.

```
request(String text) :- try{ _call(text) }
                        catch (NetworkException) {
                          slave.request(text)}
```

FIG. 9.5 – Exemple de règle ISL avec traitement d'une exception

9.4.1 Introduction d'un opérateur d'interruption

Pour nous permettre de manipuler les métaobjets au niveau des règles de réécritures nous introduisons un métaopérateur d'interruption qui permet de stopper le traitement des opérations en attente d'exécution et de modifier le métaobjet à traiter à la place de celui défini par le plan de base. Nous considérons cette opération comme une métaopération car elle n'est pas du même niveau que les opérations définies précédemment. En effet elle ne rentre pas dans le mécanisme de composition interne des règles mais modifie le séquencement des métaobjets qui sont une donnée extérieure aux règles. Ce métaopérateur se note de la manière suivante :

$$< \text{break } nomDuMétaobjet >$$

Ce métaopérateur ne se compose pas avec les autres opérateurs du langage. Lors de la composition le méta-opérateur est maintenu comme une annotation à l'endroit de sa définition. Lors de l'exécution d'une règle, avant l'exécution de chaque opérateur on vérifie s'il y a une annotation posée sur l'opérateur courant, dans le cas positif cette annotation est exécutée. L'exécution de la règle est stoppée et la main est donnée au métaobjet pointé par l'opérateur d'interruption.

Quand deux interruptions se retrouvent définies au même endroit après la composition de règles alors c'est un conflit, à moins qu'elles ne pointent sur le même métaobjet.

9.5 Le type `Message` sous-type de `Requête`

Nous détaillons ici le type `Message` qui est le type des messages qui circulent entre certains métaobjets.

$Message$:	typeRetour :	$tq\ typeRetour \in Type$
	valeurRetour :	$tq\ valeurRetour \in V(tr)$
	nom :	$tq\ nom \in Ident$
	typesParamètres :	$tq\ typesParamètres = \{t_1, ..., t_n \mid t_i \in Type\}$
	valeursParamètres :	$tq\ valeursParamètres = \{v_1, ..., v_n \mid v_i \in V(tp)\}$
	typesExceptions :	$tq\ typesExceptions = \{t_1, ..., t_n \mid t_i \in Exceptions\}$
	valeurException :	$tq\ valeurException \in V(te)$

Nous offrons sur ce type une interface qui permet de récupérer et de modifier toutes les informations contenues dans un message. Ainsi on peut modifier la valeur des paramètres, modifier la valeur de retour. On peut aussi modifier le nombre de paramètres et leur typage. Certaines modifications ont un impact sur la phase de projection. Elles sont effectuées au moment de l'évaluation des règles de réécriture. Nous donnons un exemple dans le chapitre suivant.

Conclusion

Dans ce chapitre nous avons discuté des systèmes externes de types, de propriétés, de visibilités et de règles que nous utilisons pour la projection vers les plateformes à composants présentées dans la première partie et plus particulièrement pour JOnAS.

Dans le chapitre suivant nous donnons des exemples d'intégration de services en se servant de ces systèmes externes et nous discutons de la composition et de la projection de ces intégrateurs.

CHAPITRE 10

Exemples et projection vers JOnAS

Dans ce chapitre nous détaillons quatre exemples d'intégration de services à l'aide de notre modèle complété par les systèmes externes que nous avons décrit dans le chapitre précédent. Nous montrons l'intégrateur d'un service de cache de méthodes, l'intégrateur d'un service d'authentification, l'intégrateur d'un service de cryptage et aussi l'intégrateur d'un service de persistance.

L'étude de ces exemples nous permet de mettre en œuvre notre modèle. Nous détaillons les étapes du processus avec les intégrateurs et une application simple de calcul de factorielle, nous discutons la résolution de conflits et la modification de la signature d'un message. Nous prenons JOnAS comme plateforme d'expérimentation.

10.1 Exemples d'intégrateurs de services

Nous commençons par présenter des exemples d'intégrateurs.

10.1.1 Intégrateur d'un service de cache de méthodes

Notre premier exemple d'intégrateur est l'intégrateur d'un service de cache de méthode. Le but du service de cache de méthode est de stocker le résultat d'un calcul pour éviter de refaire le calcul pour les appels qui ont les mêmes paramètres.

A la réception d'un message on demande au service de cache pour une signature donnée et les valeurs des paramètres si le résultat a déjà été calculé et stocké. Dans le cas où le résultat est dans le cache on renvoie directement le résultat. Dans le cas où le résultat n'est pas dans le cache alors on laisse se dérouler l'exécution du calcul et au retour on stocke le résultat dans le cache avec la signature et les valeurs des paramètres.

Pour intégrer ce service il faut disposer d'un composant qui serve de stockage au cache (qui implémente l'interface `Cache`). Il faut aussi modifier le comportement du composant dans lequel on intègre le service de cache. Quand un message est reçu par le composant il interroge le cache pour savoir si le message a déjà été exécuté (un objet de type `Message` est composé de la signature ainsi que des valeurs des paramètres) à l'aide de la méthode `resultOf`. Si le résultat est dans le cache, on affecte le résultat au message et l'on utilise l'opérateur d'interruption pour rediriger l'appel

vers le métaobjet `Return` ce qui court-circuite l'exécution du composant. Si le résultat n'est pas dans le cache on laisse l'appel se dérouler normalement et on modifie le comportement du métaobjet `SendBack` pour qu'il stocke le résultat de l'exécution avec la signature et les valeurs des paramètres à l'aide de la méthode `store`.

L'intégrateur figure 10.1 décrit cette intégration de service.

```
Integrator MethodCache (DAC d, MyCache implements Cache){
  [d, service metier] Cache c = new MyCache();
  [d, public metier, *, Receive(Message m)] :-
    res := cache.resultOf(m) ;
    if(res) then
      m.setResult(res)
      <break Return(m)>
    else
      _call(m)
  [d, public metier, *, SendBack(Message m)] :-
    cache.store(m) ; _call(m)
}
```

FIG. 10.1 – Intégrateur du service de cache de méthode

10.1.2 Intégrateur d'un service d'authentification

Notre second exemple est l'intégrateur d'un service d'authentification. Le but du service d'authentification est de restreindre l'accès au composant.

Dans le cas de notre service d'authentification on identifie les utilisateurs suivant le site (adresse IP) d'où ils proviennent. Le service d'authentification associe à chaque site des privilèges. Ces privilèges permettent de connaître les droits d'accès aux composants.

Pour intégrer ce service il faut disposer d'un composant qui vérifie les autorisations (qui implémente l'interface `Authentifier`). Il faut aussi modifier le comportement du composant dans lequel on intègre le service d'authentification pour pouvoir gérer les privilèges en tant que paramètres. Quand un message est envoyé au composant on demande au service d'authentification de fournir les privilèges du site sur lequel on se trouve quand on fait la demande par la méthode `getMyPrivilege`. Ensuite on ajoute un nouveau paramètre de type `Privilege` au message, et on laisse l'appel se dérouler. Quand un message est accepté on récupère la valeur du dernier paramètre qui correspond au privilège et on retire ce dernier paramètre du message. On interroge ensuite le service d'authentification pour savoir si les privilèges permettent d'invoquer le message. Dans le cas positif on laisse l'appel se dérouler et dans le cas contraire on lève une exception.

L'intégrateur figure 10.2 décrit cette intégration de service.

```
Integrator Authentication (DAC d, MyAuthentifier implements
Authentifier){
  [d, service metier] Authentifier auth = new MyAuthentifier();

  [d, public metier, *, Send(Message m)] :-
    Privilege p = auth.getMyPrivilege(this) ;
    m.addParameter("Privilege",p) ;
    _call(m)

  [d, public metier, *, Accept(Message m)] :-
    Privilege p = (Privilege) m.getParameter("Privilege");
    m.removeParameter("Privilege");
    if(auth.hasPrivilege(p,m)) then
      _call(m)
    else
      throw new PrivilegeException()
}
```

FIG. 10.2 – Intégrateur du service d'authentification

10.1.3 Intégrateur d'un service de cryptage

Notre troisième exemple est l'intégrateur d'un service de cryptage. Le but du service de cryptage est de sécuriser les données qui passent sur le réseau en les cryptant.

Pour intégrer le service de cryptage il faut disposer d'un composant qui puisse crypter un message (qui implémente l'interface `Crypter`) et d'un composant qui puisse décrypter un message (qui implémente l'interface `DeCrypter`). Il faut aussi modifier le comportement du composant dans lequel on intègre le service de cryptage. Quand un message est envoyé au composant, on demande au service de cryptage de le crypter (à l'aide de la méthode `crypt`) et on laisse l'appel se dérouler. Quand un message est accepté on le passe au service de cryptage pour qu'il soit décrypté (à l'aide de la méthode `decrypt`) et on laisse se dérouler l'appel. Notons qu'il faut que le `crypter` soit présent du coté client pour que le cryptage ait lieu avant le passage du message sur le réseau. Nous ne le gérons pas au niveau du modèle.

L'intégrateur figure 10.3 décrit cette intégration de service.

10.1.4 Intégrateur d'un service de persistance

Notre quatrième et dernier exemple est l'intégrateur d'un service de persistance. Le but du service de persistance est de stocker l'état des composants.

Ce service de persistance se configure à l'aide de deux paramètres. Le premier est le nom des attributs à sauvegarder car tous les attributs ne sont nécessairement pas intéressants à sauvegarder, et le deuxième paramètre correspond aux signatures des méthodes qui nécessitent après exécution de sauvegarder l'état du composant.

Pour intégrer ce service il faut disposer d'un composant de stockage (qui implémente l'interface `PersistantStore`). On ajoute une méthode au composant qui accède les attributs à stocker. On modifie le comportement du composant pour qu'après l'exécution d'une méthode critique il appelle la méthode `store` qui lui a été ajoutée pour qu'elle réalise le stockage.

```
Integrator Crypting (DAC d, MyCrypter implements Crypter,
MyDeCrypter implements DeCrypter){
  [d, service metier] Crypter c = new MyCrypter();
  [d, service metier] Decrypter dcr = new MyDeCrypter();

  [d, public metier, *, Send(Message m)] :-
    CryptedMessage cm = cr.crypt(m);
    _call(cm)

  [d, public metier, *, Accept(CryptedMessage cm)] :-
    Message m = dcr.decrypt(cm);
    _call(m)
}
```

FIG. 10.3 – Intégrateur du service de cryptage

L'intégrateur figure 10.4 décrit cette intégration de service. Dans cet exemple nous utilisons une syntaxe étendue pour l'intégrateur. On se sert d'éléments syntaxiques expansif semblable à un système de macro. Ici nous utilisons une boucle `for` qui permet d'itérer sur une collection d'éléments et de générer le code, issu du corps de la boucle, en remplaçant les variables, qui commencent par '$', par leur valeur. L'expansion de ces macros est effectuée en tout premier lieu pendant la création des descriptions abstraites de composants quand les paramètres sont passés aux intégrateurs.

```
Integrator Persistant (DAC d, MyDataBase implements
PeristantStore, Collection <Attribut> ac, Collection <Methode>
mc){
  [d, service metier composant] PersistantStore ps = new
  MyDataBase();

  [d, service metier composant] void store(){
  <for att in ac>
    ps.store(this, $att);
  </for>
  }

  <for meth in mc>
    [d, * metier composant, $meth, Execute(Message m)] :-
      this.store() ; _call(m)
  </for>
}
```

FIG. 10.4 – Intégrateur du service de persistance

Une autre solution pour accéder aux attributs de manière plus simple serait d'utiliser une autre configuration de métaobjets. Par exemple CODA [2] propose d'utiliser un métaobjet *Data* qui permet d'accéder aux données d'un composant. Nous pourrions ainsi récupérer les attributs suivant des propriétés du niveau méta.

10.2 Le composant Factorielle et son fichier de configuration

Nous prenons une application simple pour rester focalisé sur le processus d'intégration de service. Notre application exemple est un composant qui calcule des factorielles.

Le déployeur de l'application décide de faire tourner ce composant sur une machine très rapide. Il veut restreindre l'accès à ce composant, utiliser un cache pour éviter de recalculer plusieurs fois le même résultat, crypter les communications et rendre son composant résistant aux pannes en sauvegardant son état grâce à la persistance.

La figure 10.5 montre l'interface de notre composant factorielle.

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import java.math.BigInteger;

public interface Factorielle extends EJBObject {
    public BigInteger compute(int f) throws RemoteException;
}
```

FIG. 10.5 – Interface du composant factorielle

L'implantation du composant est donnée dans la figure 10.6.

```
import ... // Supprimé pour la concision

public class FactorielleImplBean implements EntityBean {

    // Garde la valeur en cours de calcul
    private BigInteger lastValue;

    public BigInteger compute(int f)  {
      lastValue = new BigInteger(1);
      return internalCompute(new BigInteger(f));
    }

    private BigInteger internalCompute(BigInteger b){
      if(b.compareTo(BigInteger.ONE))
        return lastValue;
      else{
        lastValue = lastValue.multiply(b);
        return internalCompute(b.substract(BigInteger.ONE));
      }
    }
```

FIG. 10.6 – Implantation du composant factorielle

Nous utilisons un appel récursif interne pour calculer la valeur de la factorielle. Cette valeur est stockée à chaque étape dans une variable d'instance de l'objet nommée `lastValue`.

Pour intégrer les différents services que nous avons vu précédemment nous définissons le fichier de configuration figure 10.7. Nous réutilisons le plan de base défini dans le chapitre 5 car il correspond au modèle RPC des EJB que nous utilisons.

Nous configurons les différents services pour qu'ils s'appliquent sur le composant factorielle. Ensuite nous utilisons un composant nommé `MyCacheImpl` pour le service de cache de méthode, nous utilisons un composant nommé MyAuthentifierImpl pour le service d'authentification, nous utilisons un composant nommé MyCrypterImpl et un composant nommé MyDeCrypterImpl pour le service de cryptage et enfin nous utilisons un composant nommé MyFileDataBaseImpl pour le service de persistance. Pour le service de persistance nous configurons la liste des attributs à sauvegarder avec comme unique membre la variable d'instance `lastValue` et nous configurons la liste des méthodes qui doivent déclencher la sauvegarde avec comme unique membre la méthode `internalCompute`.

```
<Config>
   <PlanDeBaseRPC>
      <d> Factorielle </d>
   </PlanDeBaseRPC>
   <MethodCache>
      <d> Factorielle </d>
      <MyCache> fr.essi.rainbow.MyCacheImpl </MyCache>
   </MethodCache>
   <Authentication>
      <d> Factorielle </d>
      <MyAuthentifier> fr.essi.rainbow.MyAuthentifierImpl </MyAuthentifier>
  </Authentication>
  <Crypting>
      <d> Factorielle </d>
      <MyCrypter> fr.essi.rainbow.MyCrypterImpl <MyCrypter>
      <MyDeCrypter> fr.essi.rainbow.MyDeCrypterImpl <MyDeCrypter>
  </Crypting>
  <Persistant>
      <d> Factorielle </d>
      <MyDataBase> fr.essi.rainbow.MyFileDataBaseImpl </MyDataBase>
      <ac> lastValue </ac>
      <mc> internalCompute(BigInteger) </mc>
  </Persistant>
</Config>
```

FIG. 10.7 – Fichier de configuration pour le composant factorielle

10.3 Décoration de la DAC Factorielle

La première phase du processus comme nous l'avons vu est la décoration des descriptions abstraites de composants. Dans notre exemple nous ne construisons qu'une seule description abstraite de composant pour le composant `Factorielle`. La partie structurelle est donnée dans la figure 10.8 et la partie comportementale est donnée

dans la figure 10.9. Notons que l'expansion des éléments syntaxiques expansifs a été faite avec les paramètres passés par le fichier de configuration. L'expansion du corps de la méthode de l'intégrateur de persistance n'a généré qu'une ligne de code puisqu'il n'y avait qu'une seule variable dans la liste des variables à sauvegarder et qu'il n'y a aussi qu'un seul comportement avec l'appel à la méthode `store` qui a été généré.

```
nom: Factorielle

attributs: service metier Cache c = new MyCache();
           service metier Authentifier auth = new MyAuthentifier()
           service metier Crypter c = new MyCrypter();
           service metier Decrypter dcr = new MyDeCrypter();
           metier composant PersistantStore ps = new MyDataBase();

méthodes: service metier composant void store(){
            ps.store(this, lastValue);
          }

types: 0
```

FIG. 10.8 – Partie structurelle de la description abstraite du composant Factorielle

10.4 Composition des éléments de la DAC Factorielle

Nous appliquons maintenant le processus de composition sur la description abstraite du composant Factorielle. D'abord le processus de composition traite les évolutions structurelles. Le processus lève un conflit car il y a deux variables qui ont le même nom : `service metier Cache c = new MyCache();` et `service metier Crypter c = new MyCrypter();`. Pour résoudre ce conflit le configurateur de l'application a comme possibilité soit de renommer une des variables soit de déclarer que ce n'est qu'une seule et même variable et de supprimer une des définitions. Dans notre cas ce sont deux variables distinctes qui jouent deux rôles différents, il faut donc faire un renommage. Le configurateur décide de renommer la variable `c` qui a pour type `MyCrypter` en `cr`, ce qui a un impact sur la définition d'une des évolution comportementale issue du même intégrateur. Ainsi l'appel `CryptedMessage cm = c.crypt(m)` est changé en `CryptedMessage cm = cr.crypt(m)`.

```
espacesFonctions:
    public  metier * :
        -Send(m): n := _call(m) ; Accept(n)

                + Privilege p = auth.getMyPrivilege(this) ;
                  m.addParameter("Privilege",p) ;
                  _call(m)

                + CryptedMessage cm = c.crypt(m) ;
                  _call(cm)

        -Accept(n): m := _call(n) ; Receive(m)

                +    Privilege p = (Privilege) m.getParameter("Privilege");
                     m.removeParameter("Privilege");
                     if(auth.hasPrivilege(p,m)) then
                        _call(m)
                     else throw new PrivilegeException()

                +    Message m = dcr.decrypt(cm);
                     _call(m)

        -Receive(m):  m1 := _call(m) ; Execute(m1)

                +      res := cache.resultOf(m) ;
                       if(res) then
                          m.setResult(res)
                          <break Return(m)>
                       else _call(m)

        -Execute(m):  m1 := _call(m) ; SendBack(m1)

        -SendBack(m):  n := _call(m) ; Return(n)
                  +     c.store(m) ; _call(m)

        -Return(n): _call(n)


    * metier composant BigInteger internalCompute(BigInteger) :

        -Execute(m):  this.store() ; _call(m)
```

FIG. 10.9 – Partie comportementale de la description abstraite du composant Factorielle

Dans un deuxième temps le processus compose les évolutions comportementales. Tout d'abord le processus partitionne l'ensemble des espaces de fonctions. Nous obtenons trois espaces de fonctions. Le premier est :

$$\{public\ metier\ *\} \setminus \{*\ metier\ composant\ BigInteger\ internalCompute(BigInteger)\} \tag{10.1}$$

Le deuxième est :

$$\{public\ metier\ *\} \cap \{*\ metier\ composant\ BigInteger\ internalCompute(BigInteger)\} \tag{10.2}$$

Et le troisième est :

$$\{*\ metier\ composant\ BigInteger\ internalCompute(BigInteger)\} \setminus \{public\ metier\ *\} \tag{10.3}$$

Le processus ensuite compose les règles grâce au système externe de règles. L'espace de fonctions 10.3 compose deux règles sur le métaobjet `Execute` qui nous donne la règle `this.store(); _call(m); SendBack(m1)`

La composition dans l'espace de fonctions 10.1 nous donne pour le métaobjet `Send` :

```
Privilege p = auth.getMyPrivilege(this) ;
m.addParameter("Privilege",p)  ;
CryptedMessage cm = c.crypt(m) ;
n := _call(cm) ; Accept(n)
```

Pour le métaobjet `Accept` nous avons la composition suivante :

```
Message m = dcr.decrypt(cm);
int s = m.getParameters.size() ;
Privilege p = (Privilege) m.getParameter(s);
m.removeParameter(s);
if(auth.hasPrivilege(p,m)) then
    _call(m) ; Receive(m)
else throw new PrivilegeException()
```

Pour le métaobjet `Receive` nous avons la composition suivante :

```
res := cache.resultOf(m) ;
if(res) then
    m.setResult(res)
    <break Return(m)>
else m1 := _call(m) ; Execute(m1)
```

Pour le métaobjet `SendBack` nous avons la composition suivante :

```
c.store(m) ;n := _call(m) ; Return(n)
```

Il n'y a pas de composition pour les métaobjet `Execute` et `Return` puisqu'ils ne possèdent qu'une règle chacun.

10.5 Projection

Le processus construit dans cette étape un composant concret qui représente le composant Factorielle et y introduit les attributs et les méthodes concrètes issues du composant Factorielle. Nous avons donc dans notre composant concret, un attribut concret :

```
prive metier composant BigInteger lastValue;
```

Et deux méthodes concrètes :

```
public metier composant BigInteger compute(int f)
prive metier composant BigInteger internalCompute(BigInteger b)
```

Ensuite le composant concret est enrichi par les évolutions de la description abstraite de composant. Les attributs sont ajoutés sans conflit, ainsi que la méthode puisqu'il n'y a pas de problème de nommage avec les attributs et les méthodes concrètes déjà présente.

La table de correspondance nous donne l'indication que l'attribut concret `privé metier composant BigInteger lastValue ;` est lié à l'objet d'implantation `Bean`

La méthode concrète `public métier composant BigInteger compute(int f)` reçoit les métaobjets de l'espace de fonctions 10.1 et la méthode concrète `privé métier composant BigInteger internalCompute(BigInteger b)` reçoit les métaobjets de l'espace de fonctions 10.2.

Pour la méthode concrète `public métier composant BigInteger compute(int f)` la table de correspondance nous donne les indications suivantes. Le métaobjet `Send` est lié à l'objet d'implantation `Proxy`, le métaobjet `Accept` est lié à l'objet d'implantation `Squelette`, le métaobjet `Receive` est lié à l'objet d'implantation `OI`, le métaobjet `Execute` est lié à l'objet d'implantation `Bean`, le métaobjet `SendBack` est lié à l'objet d'implantation `Squelette` et le métaobjet `Return` est lié à l'objet d'implantation `Proxy`.

Pour la méthode concrète `privé métier composant BigInteger internalCompute (BigInteger b)` le métaobjet `Execute` est lié au `Bean`.

La génération de code n'étant pas différente de celle du chapitre 5 nous ne la détaillerons pas plus avant.

Conclusion

Dans ce chapitre nous avons détaillé une application simple qui calcule une factorielle, à laquelle nous avons ajouté quatre services. Nous lui avons ajouté un service de cache de méthodes, un service d'authentification, un service de cryptage et un service de persistance. Cela nous a permis de mettre en œuvre notre modèle d'intégration de service, et d'introduire des facilités syntaxique pour l'écriture des intégrateurs (cf. Annexe).

Le pouvoir expressif des intégrateurs peut être enrichi comme nous l'avons montré dans ce chapitre, sans modifier notre modèle, grâce à des extensions syntaxiques tant que leur expansion peut se faire avant l'étape de composition.

La gestion des conflits sur les noms peut se résoudre pendant le processus de composition comme nous l'avons montré en exemple dans ce chapitre. En effet les

évolutions structurelles et comportementales sont reliées à l'intégrateur dont elles sont issues, ce qui permet de modifier leurs dépendances.

De même, la gestion des conflits des règles ISL sont résolus par l'utilisateur au moment où ils se produisent. L'utilisateur modifie la règle résultante pour convenir à son besoin.

L'opérateur d'interruption que nous avons introduis à ISL permet de casser la succession des métaobjets et nous permet donc d'interrompre le traitement des règles suivantes sur les métaobjets liés.

Nous constatons que le choix des systèmes externes influence fortement le type des évolutions permises par notre modèle d'intégration de services. Le choix de systèmes externes est difficile et contraint par notre modèle, mais permet d'étendre notre modèle de manière significative.

Récapitulatif

Dans cette partie nous avons défini un ensemble de systèmes externes qui complètent notre modèle d'intégration de services. Nous avons défini ceux-ci pour qu'ils permettent de prendre en compte les différentes plateformes à composants que nous avons exposées dans la première partie de cette thèse. Nous avons ainsi défini un système externe de type qui reprend celui de Java et y ajoute les opérations requises par notre modèle. Nous avons défini un système externe de propriétés qui propose un opérateur d'équivalence qui correspond à la notion de surcharge en Java. Nous avons défini un système de visibilités composites pour supporter les différents rôles des objets qui forment les plateformes à composants. Enfin nous avons proposé un système externe de règles basé sur les règles de réécritures ISL que nous avons étendues pour prendre en compte la notion d'interruption vers un métaobjet.

Ensuite nous avons mis en œuvre notre modèle d'intégration de services à l'aide d'un exemple d'application et quatre intégrateurs de services. Nous avons utilisé comme exemple un composant de calcul de factorielle auquel nous avons ajouté quatre services que nous avons définis à l'aide de notre modèle. Ainsi nous avons écrit un intégrateur de service de cache de méthodes, un intégrateur de service d'authentification, un intégrateur de service de cryptage et un intégrateur de service de persistance. Ce qui nous a permis de dérouler l'ensemble du processus d'intégration de services sur un exemple concret.

Conclusion générale

Dans ce mémoire de thèse, nous nous sommes intéressés à l'intégration de services dans les plateformes à composants. Notre objectif était d'offrir un support pour décrire les intégrations des services indépendamment les unes des autres et de permettre de s'abstraire des infrastructures des plateformes à composants. Après avoir évalué les différentes approches de l'intégration de services issues de la littérature, nous avons proposé un modèle qui permet de décrire des évolutions structurelles et comportementales sur des descriptions abstraites de composants. Nous avons aussi proposé un modèle pour la composition des évolutions structurelles et comportementales qui permet de définir de manière séparée les descriptions d'intégration de services et de les recomposer de manière automatique. Enfin nous avons proposé un modèle de composants concrets qui permet de projeter les intégrations de services dans les composants de l'application. Finalement nous avons exposé un exemple d'utilisation pour intégrer des services de cache de méthodes, d'authentification, de cryptage et de persistance dans une application simple de calcul.

Bilan

Pour faciliter l'utilisation de services dans une application, nous avons vu qu'il faut modifier la plateforme qui exécute l'application pour qu'elle supporte et gère les services en coordination avec l'application. L'intégration de services dans une plateforme à composants est un exercice délicat. Nous avons vu que les architectures des plateformes à composants divergent et que cela rend l'intégration d'un même service dans différentes plateformes à composants difficile, puisqu'il faut appréhender chaque nouvelle architecture.

En se basant sur ce constat nous avons défini un modèle de description abstraite de composants. Ce modèle supporte la notion d'évolution d'un composant de manière indépendante de sa plateforme d'exécution. Il permet d'une part d'ajouter des propriétés structurelles à un composant : des méthodes ou des attributs. Et d'autre part il permet des modifier les comportements des composants. Le comportement est représenté par la modélisation de l'exécution d'un message. Cette modélisation est réalisée à l'aide de métaobjets qui décrivent les différentes étapes de l'exécution du message.

Ce modèle de description abstraite de composant nous permet de nous abstraire

des implantations des plateformes à composants. Cette réponse au problème de l'hétérogénéité des infrastructures offre une solution modulaire qui conceptualise les points de jointures de la programmation par aspects. En effet ces points de jointure ne définissent pas des points de l'exécution du langage mais décrivent des points d'exécution du composant sans prendre en compte sa plateforme d'exécution. Cette approche offre un niveau de description plus abstrait qui facilite l'expression des intégrations de services.

Ensuite nous avons proposé un modèle d'intégrateur qui se base sur le modèle des descriptions abstraites de composant et qui permet de définir de manière séparé les différentes intégrations de services. Ainsi un fournisseur de services décrit dans un intégrateur les évolutions à apporter à un composant pour qu'il puisse utiliser ce service. Le paramétrage des intégrateurs se fait à l'aide d'un fichier de configuration qui est spécifique à une application.

Nous avons aussi défini un modèle de composition qui permet de détecter des conflits dans les évolutions structurelles et qui permet de composer les évolutions comportementales. Nous avons choisi d'utiliser une composition commutative et transitive qui permet une meilleure séparation des intégrateurs, puisque quelque soit l'ordre d'utilisation des intégrateurs le résultat de la composition est unique, ou génère un conflit.

Cette séparation des intégration de services, et grâce au modèle de composition, le fournisseur de services ainsi que l'utilisateur de services sont libéré de la composition des services entre eux. Le choix d'un modèle de composition associatif et commutatif permet par rapport aux approches comme la métaprogrammation et certaines approches de la programmation par aspects d'obtenir un résultat unique quelque soit l'ordre d'ajout des intégration de services.

La séparation des services permet une meilleure réutilisation des descriptions d'intégration de services. La réutilisation est renforcée grâce à le paramétrage des intégrateurs de services. Ce paramétrage est fait, à l'image de celle des plateformes à composants, à l'aide d'un fichier de configuration.

Enfin pour permettre de réaliser le lien entre le modèle des descriptions abstraites de composants et les composants, nous avons défini un modèle de composant concret qui permet de faire la projection des évolutions structurelles et comportementales dans les composants de l'application.

Ce modèle de composant concret rend effectif notre approche en permettant d'une part de vérifier que les évolutions structurelles et comportementales sont applicables à un composant donné et d'autre part à l'aide de la table de correspondance de générer le code du nouveau composant dans lequel seront intégrés les services. Cette approche est tout à fait dans la lignée de l'ingénierie dirigée par les modèles en permettant de définir l'intégration de services à un niveau abstrait et de projeter ces intégrations, une fois composées, dans des plateformes à composants concrètes. On y retrouve la notion de transformation de modèles qui dans notre cas s'applique du modèle des intégrateurs vers le modèle des descriptions abstraites de composants puis vers le modèle des composants abstraits puis enfin vers les plateformes à composants concrètes.

Nos différents modèles pour rester le plus générique possible reposent sur des systèmes externes. Nous avons montré comment ces systèmes externes permettent d'enrichir l'expressivité des intégrateurs et ainsi réduire la complexité de mise en œuvre de l'intégration de services.

Nos modèles permettent en partie de répondre aux cinq critères que nous avons définis pour qualifier l'intégration de services. Ils permettent de gérer l'hétérogénéité des plateformes à composants, de séparer les définitions des intégrations de services, de paramétrer les intégration de services, de découpler le travail du fournisseur de celui de l'utilisateur et enfin de réduire la complexité de mise en œuvre de l'intégration de services.

Pour résumer nous avons apporté :

- Un modèle qui supporte l'évolution structurelle et comportementale des composants à un niveau indépendant de sa plateforme d'exécution.
- Un modèle d'intégrateur qui permet une définition séparée des intégrations de services au niveau du modèle des descriptions abstraites de composants.
- Un modèle de composition des intégrateurs de services.
- Un modèle de composants concrets qui permet la projection des évolutions des descriptions abstraites de composants dans les plateformes à composants.

Réflexions et travaux futurs

Nous avons vu tout au long de cette thèse, certaines limites de nos modèles. Des travaux sont encore nécessaires pour améliorer ces modèles et repousser ces limitations. Nous présentons ci-dessous différents axes de recherche à approfondir.

Nouveaux opérateurs pour les évolutions structurelles

Dans notre modèle de description abstraite de composant nous avons proposé pour les évolutions structurelles la possibilité d'ajouter de nouveaux attributs et de nouvelles méthodes à un composant. Il nous semble intéressant de disposer d'une opération de retrait qui permette de retirer un attribut ou une méthode à un composant. En effet, par exemple pour des raisons d'optimisation on peut être amené à retirer des propriétés d'un composant.

L'ajout d'une telle opération de retrait implique de modifier le modèle de composition des évolutions structurelles. Ainsi il faut décider du comportement à adopter quand une propriété est ajoutée par un intégrateur et qu'un autre intégrateur veut la retirer. Il faut aussi décider de la signification de retirer une propriété qui n'est pas présente dans un composant.

Le modèle de composition des évolutions comportementales

Nous avons fait le choix d'une composition indépendante de l'ordre des déclarations d'intégration de services. Cela permet de libérer le configurateur de la connaissance de l'ensemble des services, ce qui est conforme à une approche de la programmation basée sur la séparation des préoccupations. Cependant notre approche ne permet pas certaines compositions particulières. En effet notre composition automatique présuppose qu'il n'y a pas d'effet de bord entre les différentes règles qui sont composées. Mais ce n'est pas toujours cas, surtout quand plusieurs règles accèdent à une même propriété.

Une analyse des règles de réécritures peut permettre de résoudre certains cas à effet de bord et permettre de détecter les autres cas insolubles de manière automatique.

Application « réelle » et optimisation

Dans cette thèse nous avons présenté une application extrêmement simple, avec des services qui ne nécessitent pas trop de paramétrage, ce qui a permis de dérouler le processus de manière simple. Mais l'intégration de services plus complexes dans une application « réelle » serait une validation supplémentaire de notre approche.

De plus nous pensons qu'il est essentiel de traiter le problème de l'optimisation de notre processus au moment du partitionnement des espaces de fonctions. En théorie nous pouvons avoir une explosion combinatoire du nombre d'états lors des calculs sur les expressions régulières. Il est donc essentiel d'améliorer l'élimination des espaces de fonctions vides pour éviter des calculs inutiles et donc minimiser la possibilité d'une explosion combinatoire.

Nous n'avons jamais, dans cette thèse, abordé le problème de l'efficacité. Il serait intéressant d'effectuer des mesures de performances de notre solution.

Génération de code

Nous avons proposé la notion de table de correspondance pour permettre de factoriser les relations entre les objets d'implantation et la visibilité et d'autre part entre les objets d'implantation et les métaobjets. Nous pensons que cette table peut permettre de pousser jusqu'au bout le mécanisme de la génération de code et porter les spécificités de chacun des objets d'implantation qui sont aujourd'hui induites. Par exemple comme nous l'avons vu pour le service de cryptage, elle doit pouvoir renforcer le fait que le `crypter` soit du côté client.

Nouveaux plans de base pour d'autres cibles

Dans cette thèse nous n'avons présenté en exemple qu'un seul plan de base qui représente un flot RPC. Mais d'autres protocoles de communication entre objets existent et la description de ceux-ci sous forme de nouveaux plans de base nous semble essentielle. De nombreuses questions sont soulevées par l'utilisation de nouveaux plans de base. Par exemple, les intégrateurs que nous avons définis sur un plan de base sont-ils tous réutilisable sur un autre plan de base ? Dans quelle mesure, la définition d'un intégrateur induit-elle une connaissance totale ou partielle du plan de base ?

Services web

Nous avons eu pour objectif dans cette thèse d'étudier les problèmes d'intégration de services pour les plateformes à composants. Mais les plateformes pour les services web souffrent des mêmes problèmes pour la prise en compte de nouvelles fonctionnalités. Beaucoup de travaux autour des contrats entre services (*Service Level Agreement* - SLA) sont menés actuellement [29]. Nous pensons que notre approche est transposable aux services web et serait un apport dans la prise en compte des SLA.

ANNEXE A

EBNF et prototype

Dans cette annexe nous détaillons succinctement la réalisation de l'approche de l'intégration de services présentée dans cette thèse. Nous donnons tout d'abord la grammaire EBNF des intégrateurs de services puis nous décrivons l'architecture du prototype qui implémente notre approche.

Grammaire EBNF des intégrateurs

Nous décrivons les éléments principaux de la grammaire des intégrateurs de services en laissant certains éléments de cotés. Les termes précédés d'un souligné correspondent dans notre implantation à des éléments de la grammaire Java que nous ne détaillons pas en ici.

`_Name` correspond à un identifiant dans la syntaxe Java. `_Type` correspond à une spécification de type (type de base ou type construit). `_Variable_declaration` correspond à la déclaration d'une variable construite à partir d'une visibilité Java, d'un type, d'un nom de variable et potentiellement d'une initialisation. `_Method_declaration` correspond à la définition d'une méthode Java à partir d'une signature et d'un bloc d'expressions. `_Regular_Expression` correspond à une expression régulière étendue. `_Method_invocation` correspond à un appel de méthode Java. `_Statement_block` correspond à une suite d'expression Java entre accolade.

```
%left ISLRule ;


Integrator :
        IntegratorHeader
        "{"
        Rules
        "}"
        ;

IntegratorHeader :
        "Integrator" _Name "(" List_param ")"
```

```
        ;

List_param :
        | Param
        | List_param  "," Param ;

Param :
        _Type _Name
        | _Name "implements" _Type
        ;

Rules:
        Rule+;

Rule :
        StructuralRule
        | BehavioralRule
        ;

StructuralRule :
        "[" _Name "," Visibility "]" StructuralRuleBody ;

StructuralRuleBody :
        _Variable_declaration
        | _Method_declaration
        ;

BehavioralRule :
        "[" _Name  "," Visibility "," Method_name  ","
        Meta_object  "]" BehaviorRuleBody
        ;

Method_name :
        _RegularExpression
        ;

Meta_object :
        _Name "(" Argument_list_opt ")"
        ;

Argument_list_opt :
        | Opt
        | Argument_list_opt  "," Opt ;

Opt :
        _Name
        ;
```

```
BehaviorRuleBody :
        ":-" ISLRule
        ;

ISLRule :
        _Method_invocation
        | _Statement_block
        | "<" "break" Meta_object ">"
        | ISLRule ";" ISLRule
        | ISLRule "//" ISLRule
        | "if" "(" Method_invocation ")" "then" "{" ISLRule "}"
          "else" "{" ISLRule "}"
        | "throw" _Method_invocation
        | "try" _Statement_block "catch" "(" _Type ")"
          _Statement_block
        | "wait" _Name
        | "delegate" ISLRule
        ;

Visibility :
        AccesE Role AccesD
        ;

AccesE :
        "public"
        | "service"
        | "privé"
        | "*"
        ;

Role :
        "home"
        | "metier"
        | "*"
        ;

AccesD :
        "composant"
        | "unknown"
        | "*"
        ;
```

Architecture du prototype

Le prototype est composé de plusieurs blocs à l'image du modèle. Nous disposons, en plus de tous les éléments décrit dans cette thèse, d'un parser d'intégrateurs, d'un parser XML, d'un parser Java, d'une fabrique d'intégrateurs et d'une fabrique de descriptions abstraites de composants. Ce prototype est complètement écrit en Smalltalk

et permet de générer du code Java pour la plateforme JOnAS (version 2.1). Le générateur de code Java pour la plateforme OpenCCM (version 0.6) n'est pas totalement fonctionnel mais permet de vérifier la validité des concepts. Le prototype remplace entièrement le générateur GenIC de JOnAS. Le prototype au moment de la génération de code génère donc l'ensemble des classes normalement construites par GenIC. Nous ne nous sommes intéressé dans le prototype qu'aux *session beans*.

Tout d'abord le parser permet de construire les représentations mémoires des intégrateurs. Le parser XML permet de lire le fichier de configuration et lance la construction des descriptions abstraites de composants qui sont vides au départ à l'exception de leur nom. Les descriptions abstraites de composants sont toutes stockées dans une fabrique qui permet de les appeler par le nom du composant qu'elles représentent. Chaque intégrateur, construit par le parser, est alors appelé pour venir ajouter ses évolutions structurelles et comportementales aux descriptions abstraites de composants, auxquelles il est lié par le fichier de configuration. Ensuite chaque description abstraite de composant va faire appel d'abord à l'opérateur de fusion structurelle puis à l'opérateur de fusion comportementale.

Le parser Java permet de lire le fichier Java correspondant au composant (le bean) et les interfaces liées au composant. Le parser Java construit les composants concrets. Ensuite la fabrique de descriptions abstraites de composants lie les composants concrets et les descriptions abstraites de composants. Le générateur Java fabrique les représentations en terme de classes des composants (une classe squelette, une classe proxy, une classe pour l'objet d'interposition). A l'aide de la table de correspondances et des composants concrets, ces représentations sont remplies. Le générateur Java génère ensuite les fichiers classes Java.

BIBLIOGRAPHIE

[1] Jeff Mc Affer. Meta-level architecture support for distributed objects. In *International Workshop on Object-Orientation in Operating Systems (IWOOS 95)*, 1995.

[2] Jeff Mc Affer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214. Springer-Verlag, 1995.

[3] M. Bartorello, H. Maguin, M. Blay-Fornarino, A.-M. Pinna-Dery, and M. Riveill. Adjonction de services au sein d'un serveur EJB. In *Journées composants : flexibilité du système au langage, Besançon*, 25 et 26 octobre 2001.

[4] Laurent Berger. Mise en œuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO. In *Thèse de doctorat, Equipe Rainbow, Université de Nice Sophia Antipolis*, 2001.

[5] Philip Bernstein and Eric Newcomer. *Principles of transaction processing : for the systems professional*. Morgan Kaufmann Publishers Inc., 1997.

[6] Xavier Blanc, Olivier Caron, Arnaud Georgin, and Alexis Muller. Transformations de modèles : d'un modèle abstrait aux modèles EJB et CCM. In *LMO'04*. Hermès Science Publications, 2004.

[7] M. Blay-Fornarino, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Mapping composition patterns to AspectJ and Hyper/J. *Revue TSI*, 2004.

[8] M. Blay-Fornarino, D. Ensellem, A. Occello, A-M. Pinna-Dery, M. Riveill, J. Fierstone, O. Nano, and G. Chabert. Un service d'interactions : principes et implantation. In ISBN 2-7261-1229-3 INRIA, editor, *Journée composants : Systèmes à composants adaptables et extensibles, Grenoble, France*, 17 et 18 octobre 2002.

[9] Daniel D. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *Sigplan Notices*, 23, november 1986.

[10] Erwan Breton and Jean Bézivin. Weaving definition and execution aspects of process meta-models. In *In Proceedings of the 35th HICSS-35 Minitrack Software Technology/Integrated Modeling of Distributed Systems and Workflow Applications*, Waikoloa, Hawaii, January 2002.

[11] Erwan Breton and Jean Bézivin. Model driven process engineering. In *In Proceedings of the 25th Annual International Computer Software and Application Conference (COMPSAC 2001)*, Chicago, Illinois, October 2001.

[12] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *In Proceedings of the Second International Conference on Formal Ontology in Information Systems (FOIS-2001)*, Ogunquit, Maine, USA, October 2001.

[13] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in smalltalk-80. *SIGPLAN Not.*, 24(10) :419–431, 1989.

[14] Jean-Pierre Briot. Actalk : A testbed for classifying and designing actor languages in the smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129, Nottingham, 1989. Cambridge University Press.

[15] E. Bruneton and M. Riveill. Javapod : an adaptable and extensible component platform. In *Workshop on Reflective Middleware*, New York USA, April 2000.

[16] Bill Burke, Austin Chau, Marc Fleury, Adrian Brock, Andy Godwin, and Harald Gliebe. JBoss aspect oriented programming. http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop, February 2004.

[17] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *ASE'01, Automated Software Engineering*, San Diego, USA, November 26-29, 2001.

[18] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. In *LMO'03*, pages 241–255. Hermès Science Publications, 2003.

[19] S. Chandra, J. R. Larus, M. Dahlin, B. Richards, R. Y. Wang, and T. E. Anderson. Experience with a language for writing coherence protocols. In *Proc. of the USENIX Conference on Domain-Specific Languages (DSL)*, 1997.

[20] Siobhán Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1) :71–100, July 2001.

[21] Siobhàn Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design : towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10) :325–339, 1999.

[22] Siobhán Clarke and Robert J. Walker. Mapping composition patterns to AspectJ and Hyper/J. In *International Conference on Software Engineering (ICSE 2001)*, May 2001.

[23] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE. In *ECOOP*, pages 219–243, 2004.

[24] Philippe Collet and Roger Rousseau. Contrôle d'admission de composants avec des contrats comportementaux. In *LMO 03*, pages 31–44. Hermès Science Publications, 2003.

[25] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation. International Seminar.*, pages 54–72, Dagstuhl Castle, Germany, 12-16 1996. Springer-Verlag, Berlin, Germany.

[26] IBM Corporation. SOMobjects : A practical introduction to SOM and DSOM. Redbook, 1994. GG24-4357-00.

[27] T. Coupaye, E. Bruneton, and J.B. Stefani. The Fractal composition framework. Specification, France Telecom, http ://fractal.objectweb.org, July 2002.

[28] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[29] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand : WSLA-driven automated management. *IBM Syst. J.*, 43(1) :136–158, 2004.

[30] Pierre-Charles David. Une infrastructure pour middleware adaptable. Rapport de DEA, École des Mines de Nantes, Septembre 2001.

[31] Anne-Marie Dery, Mireille Blay-Fornarino, and Sabine Moisan. Distributed access knowledge-based system : Reified interaction service for trace and control. In *3nd International Symposium on Distributed Object Applications (DOA 2001)*, Rome, Italy, September 17-20, 2001.

[32] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 02)*, pages 173–188, Pittsburgh, PA, octobre 2002.

[33] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192 :170–184, 2001.

[34] Rémi Douence and Mario Südholt. Un modèle et un outil pour la programmation par aspects événementiels. In *LMO 2003*, Vannes, February 2003. Hermès. version anglaise : [32].

[35] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1998.

[36] Samuel Eilenberg. *Automata, Languages, and Machines*. Academic Press, Inc., 1974.

[37] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98 : The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[38] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS : integrating object-oriented and functional programming. *Commun. ACM*, 34(9) :29–38, 1991.

[39] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In ACM SIGPLAN Notices 37(10), editor, *In Proceedings of the 17 Annual Conference on ObjectOriented Programming Systems, Languages, and Applications*, pages 161–173, Seattle, Washington, October 2002.

[40] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, and Miguel Toro. Introducing separation of aspects at design time. In *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.

[41] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, and Miguel Toro. Introducing separation of aspects at design time. In *In Proc. Aspects and Dimensions of Concerns workshop at ECOOP 2000*, 2000.

[42] Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau. A toolkit for weaving aspect oriented UML designs. In *AOSD 2002*, pages 99–105, 2002.

[43] Wai-Ming Ho, Francois Pennaneac'h, Jean-Marc Jezequel, and Noel Plouzeau. Aspect-oriented design with the UML. In *In Proc. Multi-Dimensional Separation of Concerns workshop at ICSE 2000*, 2000.

[44] Zahi Jarir, Pierre-Charles David, and Thomas Ledoux. Dynamic adaptability of services in enterprise JavaBeans architecture. In *Seventh International Workshop on Component-Oriented Programming (WCOP'02) at ECOOP 2002*, Malaga, Spain, June 2002.

[45] JBoss. Jboss. Implementation, JBoss, http ://www.jboss.org/, 18 septembre 2003.

[46] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[47] deRivières Jim Kickzales Gregor and Bobrow Daniel G. *The Art of Meta Object Protocol*. MIT Press, 1991.

[48] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072 :327–355, 2001.

[49] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Heidelberg, 1997.

[50] Jorg Kienzle and Rachid Guerraoui. AOP : Does it make sense ? the case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61. Springer-Verlag, 2002.

[51] Jan Kleindienst, Frantisek Plasil, and Petr Tuma. Lessons learned from implementing the CORBA persistent object service. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–167. ACM Press, 1996.

[52] Arthur H. Lee and Joseph L. Zachary. Reflections on metaprogramming. In *IEEE Transactions on Software Engineering*, volume 21, pages 883–892, novembre 1995.

[53] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations : Combining modules and aspects, 2003.

[54] A. Lopes and J. Fiadeiro. Superposition : Composition vs refinement of non-deterministic, action-based systems. *Electronic Notes in Theoretical Computer Science*, 70(3), 2002.

[55] Cristina Videira Lopes and Gregor Kiczales. D : A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox, Palo Alto, CA, USA, February 1997.

[56] Jacques Malenfant and Simon Denier. ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs. In *LMO 03*, pages 91–103. Hermès Science Publications, 2003.

[57] V. Marangozova and D. Hagimont. Non-functional replication management in the corba component model. in proceedings of the 8th international ifip/acm conference on object-oriented information systems, montpellier (france), September 2002.

[58] Raphaël Marvie. *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants*. Thèse de doctorat, Thèse de Université de Lille, décembre 2002.

[59] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar, 2003.

[60] P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, 2(3) :283–294, 1990.

[61] Microsoft. .net. Technical report, Microsoft, http ://www.microsoft.com/net/, 09 avril 2003.

[62] O. Nano and M. Blay. Annotations et transformations de modèles pour l'intégration de services. *Conférence Langages et Modèles à Objets - LM0 2004, Lille $France$ - publié dans la revue RTSI - série l'Objet (Lavoisier Eds) - ISBN : 2-7462-0887-3*, 10(2-3) :175–188, 15 au 17 mars 2004.

[63] O. Nano and M. Blay-Fornarino. Services integration by model annotation and transformation. In Paul Sammut Edited by Andy Evans and James S. Williams, editors, *First International Workshop on Metamodelling for MDA*, pages 77–92, York, England, UK, 24-25 november 2003.

[64] O. Nano and M. Blay-Fornarino. Une approche MDA pour l'intégration des services dans les plates-formes à composants. In *Journées Objets, Composants et Modèles*, Vannes, 5 février 2003. GDR ARP.

[65] O. Nano and M. Blay-Fornarino. Using MDA to integrate services in component platforms. In *Eighth International Workshop on Component-Oriented Programming (WCOP 2003) in conjuction with ECOOP2003*, Darmstat, Germany, 21 july 2003.

[66] O. Nano, M. Blay-Fornarino, A-M. Dery, and M. Riveill. An abstract model for integrating and composing services in component platforms. In *Seventh International Workshop on Component-Oriented Programming $inconjunctionwithECOOP'2002$, Malaga*, June 10 2002.

[67] Novell. Mono. Technical report, Novell, http ://www.mono-project.com, 2001.

[68] ObjectWeb. Julia. Implementation, ObjectWeb, http ://fractal.objectweb.org/tutorials/fractal/, 12 septembre 2003.

[69] ObjectWeb. Jonas. Implementation, ObjectWeb, http ://www.objectweb.org/jonas/, 30 septembre 2003.

[70] ObjectWeb. Openccm. Implementation, ObjectWeb, http ://openccm.objectweb.org/, 8 juillet 2002.

[71] Audrey Occello, Anne Marie Dery Pinna, Mireille Blay Fornarino, and Michel Riveill. Contrôle des adaptations d'applications à base de composants. In *Journées Objets, Composants et Modèles*, Vannes, 5 février 2003.

[72] OMG. Meta-object facility. Specification 2.0 ad/03-04-07, OMG, http ://www.omg.org/, 07 avril 2003.

[73] OMG. Queries / views / transformations. Qvt partners revised submission ad/2003-08-08, OMG, 08 août 2003.

[74] OMG. Security service. specification 1.8 formal/2002-03-11, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2002-03-11, 11 mars 2003.

[75] OMG. Corba 2.6 specification. Specification 2.6 formal/01-12-01, OMG, http ://www.omg.org/cgi-bin/doc ?formal/01-12-01, 12 janvier 2001.

[76] OMG. Concurrency service. Specification 1.0 formal/00-06-14, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2000-06-14, 14 juin 2000.

[77] OMG. Event service. version 1.1 formal/2001-03-01, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2001-03-01, 1er mars 2001.

[78] OMG. Life cycle service. specification 1.2 formal/2002-09-01, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2002-09-01, 1er septembre 2002.

[79] OMG. Uml 2.0 superstructure. Specification 2.0 ptc/03-08-02, OMG, http ://www.omg.org/, 2 août 2003.

[80] OMG. Naming service. specification 1.2 formal/2002-09-02, OMG, http ://www.omg.org/cgi-bin/doc ?formal/02-09-02, 2 septembre 2002.

[81] OMG. Transaction service. specification 1.4 formal/2003-09-02, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2003-09-02, 2 septembre 2003.

[82] OMG. Relationship service. specification 1.0 formal/2000-06-24, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2000-06-24, 24 juin 2000.

[83] OMG. Trading object service. specification 1.0 formal/2000-06-27, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2000-06-27, 27 juin 2000.

[84] OMG. Corba component model (ccm). Technical white paper, OMG, http ://www.omg.org/, 30 janvier 2002.

[85] OMG. Notification service. specification 1.0.1 formal/2002-08-04, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2002-08-04, 4 juillet 2002.

[86] OMG. Persistent state service. specification 2.0 formal/2002-09-06, OMG, http ://www.omg.org/cgi-bin/doc ?formal/2002-09-06, 6 aout 2002.

[87] OMG. Mda. Guide version 1.0.1 omg/03-06-01, OMG, http ://www.omg.org/mda/papers.htm, juin 2003.

[88] Robert Orfali, Dan Harkey, and Jeri Edwards. *The essential distributed objects survival guide*. John Wiley & Sons, Inc., 1995.

[89] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3) :179–202, 1996.

[90] A. Paepcke. Pclos : a flexible implementation of CLOS persistence. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 374–389. Springer-Verlag, 1988.

[91] A. Paepcke. Pclos : a critical review. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 221–253. ACM Press, 1989.

[92] A. Paepcke. *Object-Oriented Programming : the CLOS perspective*. MIT Press, 1993.

[93] M. Pasin, T. S. Weber, M. Didonet del Fabro, and M. Riveill. A highly-available replicated component-based distributed architecture. In ISBN 2-7261-1264-1 INRIA, editor, *3ème Conférence Française sur les Systèmes d'Exploitation - CFSE*, pages 549–560, La Colle sur Loup, France, 14-17 octobre 2003.

[94] R. Pawlak. *La Programmation par Aspects Interactionnelle pour la Construction d'Applications à Préoccupations Multiples*. PhD thesis, CNAM, 13 décembre 2002.

[95] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC : A flexible and efficient solution for aspect-oriented programming in java. In *Reflection 2001, LNCS 2192*, pages 1–24, September 2001.

[96] Frédéric Peschanski and Jean-Pierre Briot. Adaptation dynamiques et orthogonales de composants logiciels distribués. In *A paraître dans le numéro spécial de TSI sur l'adptation des composants*, 2004.

[97] E. RENAUX, O. CARON, and J.M. GEIB. Chaîne de production de systèmes à base de composants logiques. In *LMO'04*. Hermès Science Publications, 2004.

[98] Ed Roman, S.W. Amber, and T. Jewell. *Mastering Enterprise JavaBeans, second edition*. John Wiley and Sons, Inc., 2002.

[99] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits : Composable units of behavior. In *Proceedings ECOOP 2003*, number 2743 in LNCS, pages 248–274. Springer Verlag, jul 2003.

[100] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects : an extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37. ACM Press, 2003.

[101] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, Inc., 1999.

[102] P Stevens. Implementation of the CORBA event service in java. Master's thesis, Trinity College Dublin, Avril 1999.

[103] Sun. Entreprise java bean. Specification, Sun, http ://java.sun.com/products/ejb/, 3 juin 2003.

[104] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with aspects : Aspect support in the design phase. In *ECOOP Workshops*, pages 299–300, 1999.

[105] Clemens Szyperski and Stephan Murer. *Component software : beyond object-oriented programming 2nd ed*. Addison-Wesley, 2002.

[106] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000 (International Conference on Software Evolution)*, pages 157–167. IEEE, 2000.

[107] Mathieu Vadet and Philippe Merle. Les conteneurs ouverts dans les plates-formes à composants. In *Journées Composants*, Besançon, 25-26 octobre 2001.

[108] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : An annotated bibliography. *SIGPLAN Notices*, 35(6) :26–36, 2000.

[109] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*. Accepted for publication, June 2004.

[110] Mitchell Wand. A semantics for advice and dynamic join points in aspect-oriented programming. *Lecture Notes in Computer Science*, 2196 :45–??, 2001.

UN MODÈLE DE RÉÉCRITURE POUR L'INTÉGRATION DE SERVICES

Résumé

Les plateformes à composants visent à développer des briques logicielles réutilisables. Ces briques logicielles contiennent le code métier de l'application tandis que la plateforme d'exécution se charge de fournir et gérer le code technique (authentification, transactions, persistance, notification etc.). Le besoin croissant des applications en terme de nouveaux services fait émerger un nouvel acteur : le fournisseur de services. Son rôle est d'intégrer de nouveaux services dans les plateformes à composants.

Le fournisseur de services doit faire face à l'hétérogénéité des plateformes à composants, à la complexité des générateurs qui produisent le code technique et à la composition des différents services. Il ne bénéficie d'aucun support pour intégrer de manière homogène un service dans différentes plateformes à composants ni pour composer les services.

La première partie de cette thèse se consacre à l'étude bibliographique de l'intégration de services. Nous analysons l'intégration de services dans les intergiciels de communication, et dans les plateformes à composants. Nous discutons de la difficulté pour le fournisseur de services d'intégrer de nouveaux services dans les plateformes à composants et explorons les solutions possibles au travers des outils de séparation des préoccupations et de modélisation. Nous soulignons les lacunes de ces approches face aux problèmes de l'intégration de services dans les plateformes à composants

Dans la deuxième partie de cette thèse nous proposons un modèle abstrait de description des intégrations de services. Il permet d'une part une définition séparée des intégrations de services et d'autre part la vérification et la composition des intégrations de services en fonction de l'application. Il est constitué d'un modèle de composants abstraits qui définit des opérations pour l'évolution structurelle et comportementale des composants.

Ce modèle supporte une composition déterministe permettant de séparer les descriptions d'intégration des différents services. Le mécanisme de composition automatique et commutative repose, d'une part sur un système de réécriture pour la composition des évolutions comportementales, et d'autre part sur l'utilisation d'opérations ensemblistes pour la composition des évolutions structurelles. Ce mécanisme de composition permet de détecter des incompatibilités entre les intégrations de services. La projection dans les plateformes à composants se définit à l'aide d'un modèle intermédiaire de composants concrets qui permet de combiner les évolutions des intégrations de services et des informations issues de la plateforme à composants cible. La projection permet de détecter des incompatibilités entre les intégrations de services et la plateforme cible.

La troisième partie de cette thèse concrétise le modèle de composants abstraits et discute l'application de notre modèle d'intégration de services à l'aide d'exemples de services issus de la bibliographie.

Mots-clés : Plateformes à composants, intégration de services, ingénierie dirigée par les modèles, composition.

UN MODÈLE DE RÉÉCRITURE POUR L'INTÉGRATION DE SERVICES

Résumé

Les plateformes à composants permettent de développer des briques logicielles réutilisables. Ces briques logicielles contiennent le code métier de l'application tandis que la plateforme d'exécution se charge de fournir et gérer le code technique (authentification, transactions, persistance, notification etc.). Le besoin croissant des applications en terme de nouveaux services fait émerger un nouvel acteur : le fournisseur de services. Son rôle est d'intégrer de nouveaux services dans les plateformes à composants.

Le fournisseur de services doit faire face à l'hétérogénéité des plateformes à composants, à la complexité des générateurs qui produisent le code technique et à la composition des différents services. Il ne bénéficie d'aucun support pour intégrer de manière homogène un service dans différentes plateformes à composants ni pour composer les services.

Dans cette thèse nous proposons un modèle d'intégration de services indépendant des plateformes à composants qui permet de décrire de manière abstraite l'intégration de services. Ce modèle supporte un système de composition automatique des intégrations de services qui permet de détecter des conflits d'intégration et un processus de projection des intégrations de services dans les différentes plateformes à composants.

Mots-clés : Plateformes à composants, intégration de services, ingénierie dirigée par les modèles, composition.

A REWRITING MODEL FOR SERVICES INTEGRATION

Abstract

Component platforms enable the development of reusable pieces of software. These software pieces contain the business code while the execution platform provide and manage the technical code (authentication, transactions, persistency, event etc.). As applications need more new services, a new actor appear : the services provider. He is in charge of integrating new services in components platforms.

The services provider must face component platforms heterogeneity, complexity of generators who generate the technical code and services composition. He doesn't benefit from any support to integrate a service homogeneously in different component platforms nor to compose services.

In this thesis we propose a model for services integration which is independent of component platforms and enables the description of services integration in an abstract way. This model offers an automatic composition system for services integrations which helps the detection of integration conflicts and a system for casting services integration in various component platforms.

Keywords : Component platforms, services integration, model driven engineering, composition